BRUCE PEEL SPECIAL COLLECTIONS LIBRARY

UNIVERSITY OF ALBERTA LIBRARY

REQUEST FOR DUPLICATION

I wish a photocopy of the thesis by

_____(author)

entitled_____

The copy is for the sole purpose of private scholarly or scientific study and research. I will not reproduce, sell or distribute the copy I request, and I will not copy any substantial part of it in my own work without permission of the copyright owner. I understand that the Library performs the service of copying at my request, and I assume all copyright responsibility for the item requested.

| Date | Name and address | Pages copied | Signature |
|---|---|---|---|
| MAR - 4 1987 | Paul Gadreau<br>478 CAB | All | ILL-ML |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

The University of Alberta

# A METHODOLOGY FOR THE EVALUATION OF DATAFLOW COMPUTER ARCHITECTURES

by

Donna J. Fremont

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Fall, 1983

## ABSTRACT

Dataflow computer architectures are a new class of architectures which employ parallel processing and are based on a data driven computation model. Although these novel architectures promise increased power and speed, they complicate an already difficult task, the design of computer systems. The development of tools to manage and simplify the design process is important because data flow architectures are both untried and inherently more complex than traditional uniprocessor computers.

Presented here is a methodology used to investigate properties of proposed dataflow architectures based on their estimated performance. Fundamental to the methodology are a formal language to describe the architecture and a data-driven simulator to generate performance statistics. The methodology is tested on two dataflow architecture designs taken from the current literature, showing sufficient experimental results to demonstrate its applicability to divergent designs within the class of architectures. This work also contributes to the $S_A^*$ (an architecture description language) design environment by extending its use to the class of dataflow computer architectures.

**Acknowledgements**

I would like to thank my supervisor, John Tartar, for his constant guidance and support throughout this research. Jeff Sampson, Alan Wagner and Lisa Higham read a rough draft of the thesis and contributed helpful suggestions for improvements. Steve Sutphen assisted in the use of text and picture processing facilities during the production of this thesis. Thanks to Darrel Makarenko for his previous work on the simulator and his help during the programming of the modifications.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

# CHAPTER 1

## Introduction

### 1.1. The Problem

Computer architecture design is a complex and expensive process. Multiprocessor computer systems are now being used to experiment with a variety of homogeneous and heterogeneous processing units, allowing simultaneous use of many resources. Dataflow computer architectures are a newer class of architectures which utilize the multiprocessor concept and are based on a data driven computation model. Although these new architectures promise increased power and speed, they complicate an already difficult task, the design of new computer systems. These architectures are both untried and inherently more complex than the traditional uniprocessor computers. What tools can be developed to manage and simplify the design process of such complex and non-traditional machines?

The increased complexity of computer systems demands a systematic and structured approach to design, and requires the use of automated tools. One desired element in the computer architecture design process is the evaluation and comparison of different proposals. Architects wish to investigate the properties of proposals based on estimated performance.

### 1.2. Proposed Methodology

The goal of this thesis is to contribute towards a methodology for the evaluation and comparison of dataflow computer architectures. The proposed methodology

(1) identifies parameters and performance measures appropriate to the class of dataflow computer architectures,

(2) describes the candidate architecture using a formal language, and

(3) performs a deterministic simulation of the candidate architecture.

Evaluation implies the selection of a set of performance measures and a set of parameters, both appropriate to the common characteristics and to the design goals of the class of architectures under investigation. Since the major goal of a dataflow architecture is an increase in processing power through the exploitation of parallelism, the quantifiers must measure achieved parallelism and lost processing power due to increased communication activity. Since evaluation can be conducted only if the architecture designs are rigorously and unambiguously specified, the architecture is described using a formal language.

A deterministic simulation requires the selection of appropriate test programs (coded in machine language) and a simulation of the execution of these test programs. Input to this simulation will be the architecture description, test program code and parameter values. Output variables will be used to calculate defined performance measures. A data-driven simulation, along with a report generator and test recording facility, forms an automated environment for the specification and evaluation of proposed dataflow architectures.

This thesis attempts to validate the proposed methodology by applying it to the evaluation of two dataflow architecture proposals described in the current literature.

## 1.3. Past Work

Performance measures used to evaluate traditional uniprocessor computer architectures [FuB77] are not appropriate to dataflow architectures. Two studies [GoT80, Pla76] of proposed dataflow architectures identify relevant quantifiers, some of which are included in the set of measures used in this methodology.

There are different approaches to evaluation of architectures: analysis, simulation and construction. Construction has been used to evaluate dataflow architectures [DBL80], but is expensive and therefore cannot be used to investigate decisions such as one thousand versus one hundred processing units. Construction is not feasible until late in the design process. Analytical studies have

been performed to evaluate dataflow architectures [Jen81, Mey76, Mis76]. Analysis is difficult for complex and irregular systems and cannot react to specific input data. Simulation at the architecture level cannot predict actual performance and cost but can be used to study throughput of a system [DBL80]. Simulation is a very flexible tool applicable at many levels throughout the design process. Data-driven simulation allows experimentation with the machine instruction set as well as system structure and behavior.

Probably the most complete data-driven simulation facility is the ISPS [Bar79] architecture simulation facility at Carnegie-Mellon University. It is part of a large automated design environment which has been used for the comparison, evaluation and design of computer architectures [BaS77, VBH81]. The ISPS facility has not been used for dataflow architectures.

Gostelow and Thomas report results of experiments on a simulated version of a particular dataflow architecture executing programs written in a high level dataflow language called Id [GoT80]. The simulator is custom built for this particular machine and thus cannot be used to evaluate other designs.

A data-driven simulation facility, based on the architecture description language $S_A^*$ [Das81] has been developed at this university. $S_A^*$ is one member of a family of languages which attempts to cover the different levels of abstraction possible in architecture descriptions. Since $S_A^*$ can describe asynchronous concurrent processes, a few restrictions on the use of constructs permit its application to dataflow architectures. This research uses the $S_A^*$ language and simulator (with modifications and extensions), as the core of an experimental environment for the evaluation of dataflow architectures.

### 1.4. Outline of Thesis

In order to define the problem further, Chapter 2 discusses in more detail dataflow architectures. Besides a description of the dataflow model of computation, resultant implications for dataflow architectures are discussed and dataflow architecture proposals from the current research

are surveyed.

Chapter 3 proposes a methodology for evaluation of dataflow architectures. The components of the method are discussed in turn, first the choice of performance measures and parameters, second the use of a formal language for architecture description and finally the data-driven simulation facility.

The practicality of the methodology is tested and assessed in Chapter 4, where two dataflow architecture proposals from the literature are evaluated.

Concluding remarks are presented in Chapter 5.

# CHAPTER 2

## Dataflow Computer Architecture

### 2.1. Design Goals

The demand for higher performance in computer systems has in the past been met by increasing the speed and bandwidth of uniprocessor computer systems. Speed has been increased through design solutions such as instruction pipelining, I/O channels and cache memories. Further increases in speed are becoming more difficult to achieve and manufacturers are seriously considering alternative machine architectures. Jean-Loup Baer expresses the opinion,

> In order to achieve computational rates of the order of 100 Mflops with adequate precision, the architecture of supercomputers will have to depart from the strict von Neumann concept. ... Therefore, the decomposition of computations into tasks which can be executed concurrently will be mandatory. [Bae80]

Parallel processing computer systems, including multiprocessor and distributed systems, are a recognized solution to the search for more computing power. Unfortunately a multiprocessor system constructed simply by connecting multiple von Neumann uniprocessors encounters difficult problems where synchronization is concerned.

New VLSI technology is promising greater speed, reduced cost, increased reliability and reduced power consumption; therefore the constraints of VLSI technology must be considered when designing new architectures. Because of the high design cost of VLSI chips, an economic advantage is attained only if a chip is produced in large volumes. This is achieved either if the chip is widely used in a variety of products or if one computer system incorporates many identical chips. These economic factors encourage the design of computer systems using many identical complex-logic chips connected together in simple, regular structures. Traditional uniprocessor designs do not exhibit these features, hence more emphasis is being placed on multiprocessor designs.

John Backus [Bac78] writes forcefully of the problems inherent to conventional programming languages based on the von Neumann architecture. Besides being large and inflexible, procedural languages are difficult to prove (involve side effects) while parallel procedural languages are even worse. In short, "conventional languages create unnecessary confusion in the way we think about programs" [Bac78,p. 614]. By contrast, functional languages are simpler, more powerful and possess mathematical properties which can be used to prove programs. Unfortunately, the main drawback has been that implementation of functional languages on von Neumann architectures is grossly inefficient. Therefore Backus concludes that alternate architectures specifically designed for functional languages should be developed. Since dataflow computation is functional by nature, dataflow architectures would directly support functional languages.

The above mentioned trends in computer design are consistent with current interest in the dataflow computer as an alternative to the von Neumann uniprocessor. Research is presently being conducted in the United States, Great Britain, France and Japan. The goals of dataflow architecture as seen by these researchers are:

(1) increased computer performance through concurrency;

(2) exploitation of VLSI through a computer organization consisting of identical complex functional units connected together in a regular structure with little off-chip communication;

(3) direct support of functional programming languages resulting in increased reliability due to easier verification of functional programs.

Research in Japan towards *Fifth Generation Computer Systems* is concentrating on four topics: knowledge-based expert systems, very-high level programming languages, decentralized and parallel computing, and VLSI technology [TrL82]. Knowledge-based expert systems will require high computer performance. It is likely that a dataflow or reduction architecture will be employed in the Fifth Generation computers as they directly support functional programming languages and pursue higher performance through exploitation of concurrency and VLSI.

## 2.2. Dataflow Computing Model

A computer system is based on an underlying computing model. The components of the system, namely programming languages, operating system and machine architecture, maintain consistency with one another because they are all based on the same computing model [DaD80].

The von Neumann computer architecture is the most prevalent computing model on which computer systems are based. Work began in 1946 at Harvard University to design the IAS computer. It is now considered the prototype of all subsequent general purpose computers [Hay78]. John von Neumann headed the design team of the IAS, hence the term *von Neumann architecture* is implicitly defined by the characteristics of the IAS computer. Those characteristics are:

(1) Instructions *and* data are stored in main memory (stored program concept).

(2) The computer is organized with one main memory and one central processing unit (CPU). A fixed number of bits form a word (IAS had forty bits to one word) and one word at a time is transferred between main memory and the CPU.

(3) The CPU initiates all control signals which are synchronized by a central clock.

(4) The instruction cycle consists of two consecutive steps: instruction fetch and instruction execute.

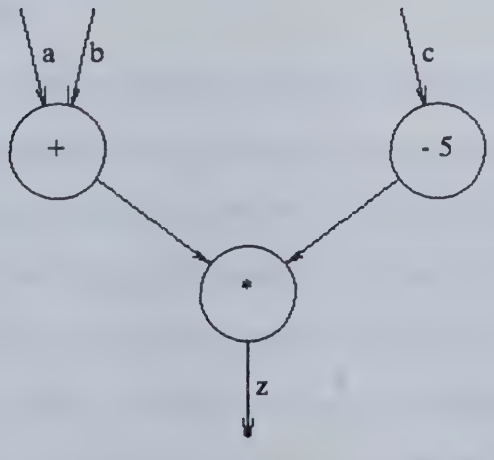
(5) An instruction does not explicitly contain the address of the next instruction to be executed. Unless otherwise directed the next instruction to be executed is the next sequential instruction of the program. A program counter maintains the address of the next instruction and this program counter is one of a number of registers within the CPU.

(6) Two's complement arithmetic is used.

(7) Instructions are composed of an opcode and an operand address.

Computer systems have certainly advanced since the IAS but the seven characteristics cited above remain the basis of most computer systems.

The dataflow computing model [DaD80, TBH82] is derived from a program representation, not from a computer architecture as is the von Neumann model. A dataflow program can be mapped to a directed graph where the nodes represent operations to be performed and the arcs represent the data paths between nodes. A node (operation, activity) acts (fires) only when each of its input arcs contains data values (tokens). On firing of a node, all input tokens are destroyed and a result token is produced and placed on all outgoing arcs. An arc operates as a first-in-first-out queue, delivering one data token at a time to the consuming node. A node may have any number of input and output arcs. All data flows forward in the directed graph and each node fires when all of the data required by its input arcs is available. It is not possible to determine the absolute order of node firing; but nodes requiring input data tokens produced by other nodes always fire after the latter. Therefore the sequencing required is solely determined by data dependencies within the program. An example of a computation represented as a directed graph is shown in Figure 2.1.




$z \leftarrow (a + b) * (c - 5)$

**Figure 2.1 Computation as a Directed Graph**

### 2.3. Implications of the Dataflow Model

This abstract dataflow computing model forms the basis for the design of dataflow computer architectures. Computer architecture may be informally defined as the structure and behavior of a computer, but a computer architecture can be examined at many levels from the detailed circuit-level to the processor-memory-switch (PMS) level of Bell and Newell [BeN71]. An appropriate abstraction level during the initial design process is the endo-architecture level.

> Endo-architecture typically includes the functional capabilities of a machine's physical components, their interconnections, the nature of the information flow between components, and the means whereby this flow is controlled. [Das81]

The following discussion considers computer architecture at the endo-architecture level and is organized into three topics similar to those examined by Treleaven [TBH82]: program organization, execution cycle and machine organization. Program organization includes the representation of machine language programs. The execution cycle describes the sequencing and results of execution. Machine organization refers to the configuration of machine resources and their allocation to support program organization.

#### 2.3.1. Program Organization

In the von Neumann model a variable is synonymous with a storage location, and an operation normally involves fetching data values, performing a computation and updating a storage location. In contrast, dataflow data values are not stored, but exist only as tokens in transit from producer to consumer node. Theoretically, the value of a variable is never updated, therefore high level dataflow programming languages uphold a single-assignment rule whereby a variable can be assigned a value only once. With no updating possible, synchronization of access to variables is not a problem, and all instructions which are not data dependent can be executed simultaneously thus allowing concurrent computation within one process.

As well as a value, a data token must contain the name or address of its destination instruction along with other information such as processor address, unique code block identifier, iteration

number, argument number, number of arguments required and the name of the producing instruction. Therefore data tokens require more space in transit than the equivalent data value would require in the memory of a von Neumann machine. If dynamic concurrency as well as static concurrency[1] is desired then more information is required to insure uniqueness of destination addresses.

Results of computations can be used as input by more than one instruction, hence a machine language instruction must include names and information for all destinations. Other information required in instructions includes: opcode, opcode dependent information (decision flags, alternate routes), value of a constant and operand number, number of destinations, number of operands, operand flags, acknowledgement signals, number of acknowledgement signals required. As indicated, dataflow instructions are complex and lengthy.

Since dataflow programs do not explicitly control the sequence of execution, high level language programs are simpler to construct. Data flow program organization is very efficient for evaluation of simple expressions and support of procedures and functions with call by value parameters [TBH82]. On the other hand dataflow program organization is not good at manipulation of shared data structures.

### 2.3.2. Execution Cycle

Logically, every instruction in a data flow program is active and has a processing element allocated to it, waiting for arguments to arrive. When all arguments have arrived the computation is performed, the arguments are destroyed and the result sent to each of its destinations. The processing element then becomes idle. If all instructions are data-independent, then theoretically all instructions could execute simultaneously. No control information is needed because execution is initiated asynchronously upon the arrival of data. No program counter is needed because all

[1]Static concurrency refers to the concurrency made possible by data-independence. This concurrency is evident at compile time. Dynamic concurrency refers to the concurrent execution of different loop iterations or function invocations. Dynamic concurrency is determined during execution.

sequencing is implicit through data dependency. A result token is the only result of execution. It would appear then that the execution cycle of a dataflow architecture is extremely simple and since no clock or synchronized activity is required a distributed environment is possible.

### 2.3.3. Machine Organization

As previously stated, each instruction is in principle active with a processing element allocated to it. In reality, it would be wasteful of computing resources to allocate one processing element to each instruction. Rather, the essential task of a dataflow computing system is to find executable instructions and then to allocate them to the appropriate resources.

Treleaven describes two ways of solving the problem of finding an executable instruction: token storage and token matching. In token storage the values of the operands are stored with the instruction in memory. Production of a result token will cause an update of all destination instructions (i.e. the program itself is changed). Some mechanism must search memory to find complete instructions and fetch them to a processor for execution. Alternately, token matching requires a separate memory for result tokens. Each new result token is matched up with other tokens bound for the same destination. When a set of instruction operands is complete, the set of operands is released and activates execution of that instruction. Program code is not modified as it is in token storage, therefore token matching supports reentrant code.

The asynchronous concurrent nature of dataflow programs suggests a computer system composed of more than one processor, where each processor acts independently and the whole system has no central control. A distributed system of independent processors can be realized in many ways. If each processor is specialized then a communication network is needed to route an instruction to the appropriate processing element. If all processing elements are identical then an idle processor must be found.

Instead of each processor performing all decision making and computation, it is possible to separate the processor into many units each with a particular function to perform within the

instruction cycle. Some functions to be performed are: token matching, insertion of operands into instructions, determining if an instruction is executable, fetching instructions, composing operation packets, performing computation and constructing tokens from results. Separate units performing different functions on different instructions can operate concurrently.

### 2.3.4. Summary

The data flow computing model suggests certain characteristics in a data flow architecture. The independent concurrent nature of instructions implies many processors (either identical or specialized) performing arithmetic and logical computations. The asynchronous aspect of the execution cycle suggests specialized functional units (e.g. token matching, token production, token routing, memory updating) operating concurrently and communicating with each other asynchronously. Since the results of computation are local and memory need not be shared, distributed control and memory are possible. Therefore a dataflow architecture is likely to be a multiprocessor system with asynchronous communication and distributed control.

More specifically, the representation of programs must be concerned with:

(1) the set of primitive operations (e.g. arithmetic, logical, data structure manipulation, token selection and routing);

(2) machine language instruction format;

(3) data token format;

(4) use of control tokens;

(5) appropriate high level languages and their translation to machine level instructions.

The structure of the architecture must reflect decisions concerning:

(1) functional units required;

(2) identical or specialized functional and computing units;

(3) communication paths between units;

(4) distribution of program and data storage.

The behavior of the architecture must solve the following problems:

(1) determining if a node is ready to fire (token storage and token matching are two current solutions);

(2) mapping of instructions to computing elements and programs to storage;

(3) dynamic node replication.

## 2.4. Survey of Dataflow Architectures

Researchers are currently investigating the feasibility of the dataflow computing model. Many dataflow architectures have been proposed. The following is a brief description of five proposals, which are often cited and reveal the variety of approaches to the problem. Both Treleaven [TBH82] and Davis [DaD80] have produced comprehensive surveys of the current research.

### 2.4.1. MIT Machine

Extensive dataflow research has been conducted at the Massachusetts Institute of Technology (MIT) Computer Structures Group under Jack Dennis. Topics of research include high level dataflow programming languages and translators, decomposition of programs, architecture, routing networks and logic simulators [Mas80].

The Dennis/Misunas proposed dataflow architecture [DeM75, DLM80] is composed of a memory subsystem, a operation subsystem and three communications networks (see Figure 2.2). The memory subsystem stores instructions and their operands in instruction cells. The operation subsystem is composed of special purpose processing elements which perform arithmetic and logical computations. The arbitration network is a switching network which receives executable instruction packets from the memory subsystem and sends them to the appropriate processing element in the operation subsystem. The distribution network is a switching network which receives data

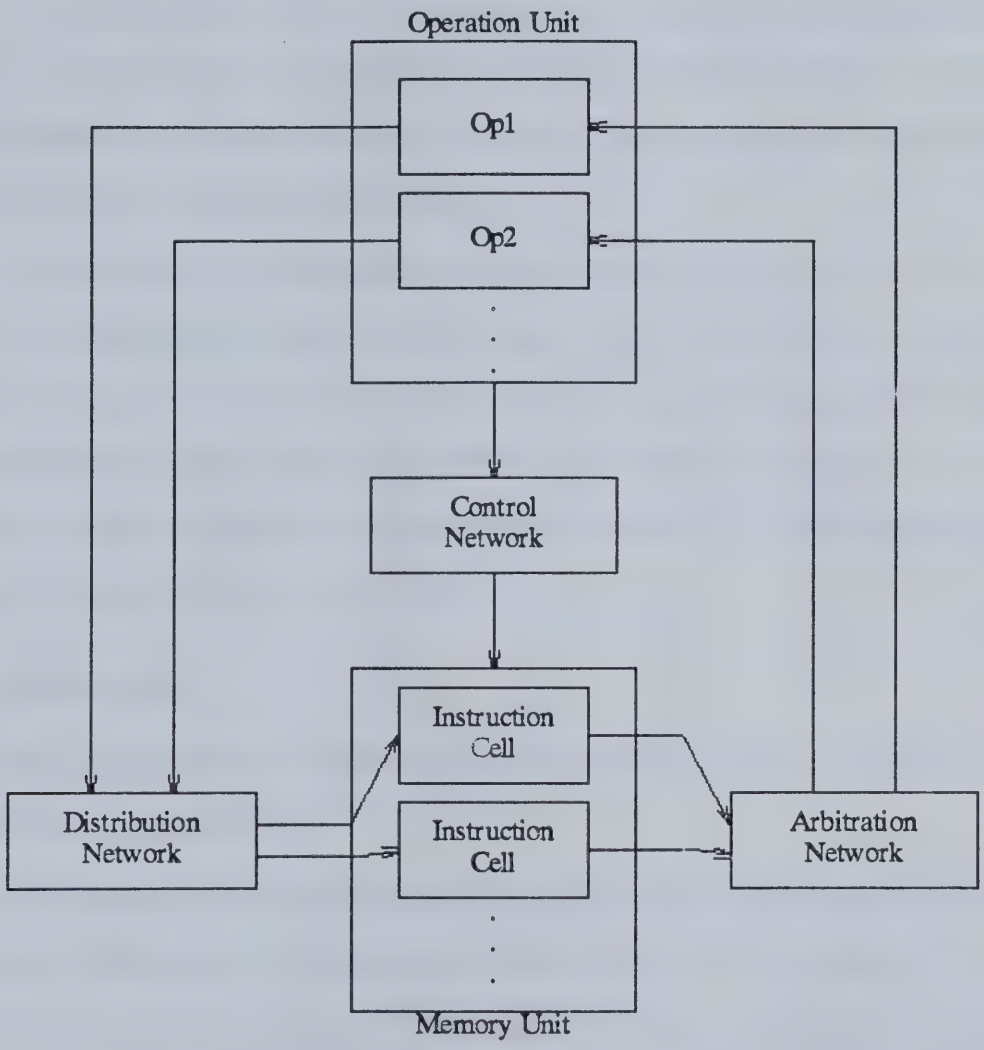



**Figure 2.2 MIT Machine**

packets and sends them to the memory subsystem. The control network performs a function similar to the distribution network, except that the control packets are either data values of type boolean bound for conditional or iterative control instructions or are acknowledgement signals sent to producing instructions. The memory subsystem receives data and control packets through the respective networks, and stores these into appropriate instruction cells as they arrive. If the receipt of a token makes an instruction executable, an instruction packet is created and sent to the operation unit through the arbitration network. All communication between the five units is

asynchronous.

The structure of this dataflow architecture introduces parallelism by allowing each cell in memory concurrent access to the operation unit through the arbitration network. Parallelism is also introduced by the specialized processing elements in the operation subsystem which allow concurrent execution of different primitive operations.

A unique feature of the MIT machine is the firing rule. A node fires when it has received its complete specified firing set and *its output arc is empty*. Therefore each arc holds only one token, an arc is not a queue. If an instruction cell has received its complete firing set and it has received an acknowledgement signal signifying consumption of its previous result token, then it is executable and will send an instruction packet to the arbitration network. The acknowledgement signal is necessary to enforce the one token per arc rule.

### 2.4.2. DDM1 Machine

The DDM1 (Data-Driven Machine #1) project resides at the University of Utah under the direction of A.L. Davis [DaD80].

This machine is a tree-structured multiprocessor composed of identical processors each having up to eight child processors. Communication between processors is asynchronous. No control tokens are used in the system and arcs act as first-in-first-out queues. A processor (see Figure 2.3) is composed of an atomic storage unit (stores instructions and operands), an atomic processor (performs operations), an agenda queue (stores messages), an input and output queue (communicates with the parent processor) and a switch (communicates with child processors).

Program instructions are allocated to processors based on the following principle. A processing element receives a subprogram from its parent. If the subprogram cannot be further decomposed, it is stored in the local atomic storage unit. Otherwise the subprogram is divided and sent to the child processors.

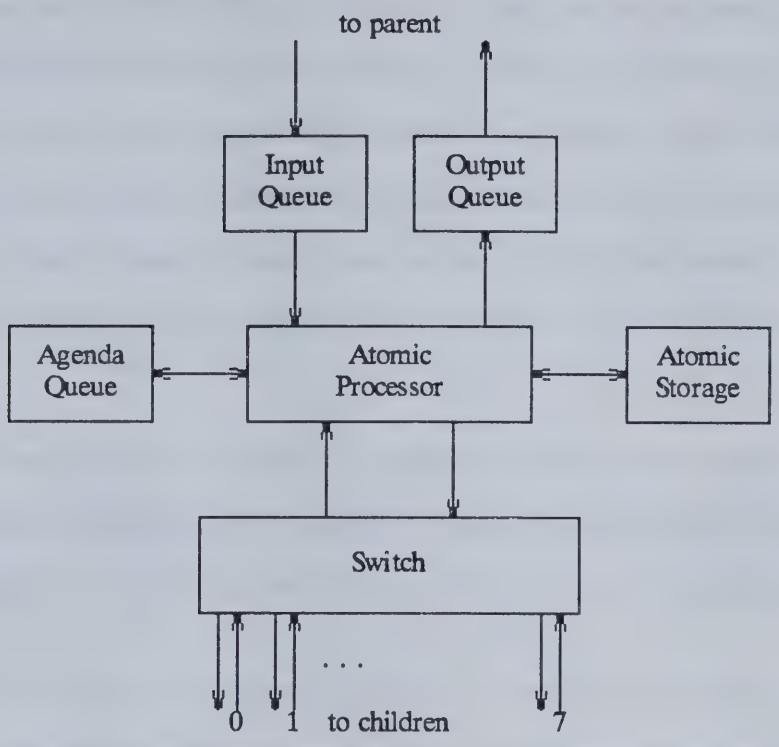



**Figure 2.3 DDM1 Machine**

The atomic processor performs arithmetic, list structuring and routing operations. The atomic processor takes messages from the switch, the agenda queue and the input queue in that order of priority. A received data token is either routed to another processor or if its destination instruction is in the local atomic storage unit it is inserted into the instruction. If the instruction is executable, the appropriate operation is performed and the result token is placed in the output queue, switch or agenda queue. Concurrency is achieved through identical processing elements.

A unique feature of this architecture is its hierarchical structure. Data tokens are list structures and storage is organized as a list structured file. Also, programs are decomposed to subprograms, not to individual instructions.

### 2.4.3. Id (Irvine Dataflow) Machine

The Id machine [AGP78, GoT80] is composed of many identical processing elements which communicate through two token buses (see Figure 2.4, the processing elements are labeled Pe0, Pe1, ...). An arbitrary number of processing elements (4) are grouped together to form a physical domain. As well as processing elements, the physical domain contains a memory controller and memory store. Each processing element communicates with its local memory, but all local memories are connected by a global memory bus. The address space of the whole system is unified.

Each processing element (see Figure 2.5) consists of special purpose functional units which communicate asynchronously through data tokens. There are no control tokens. The token buses act as shift registers and rotate in opposite directions. Each processing element accesses one slot of

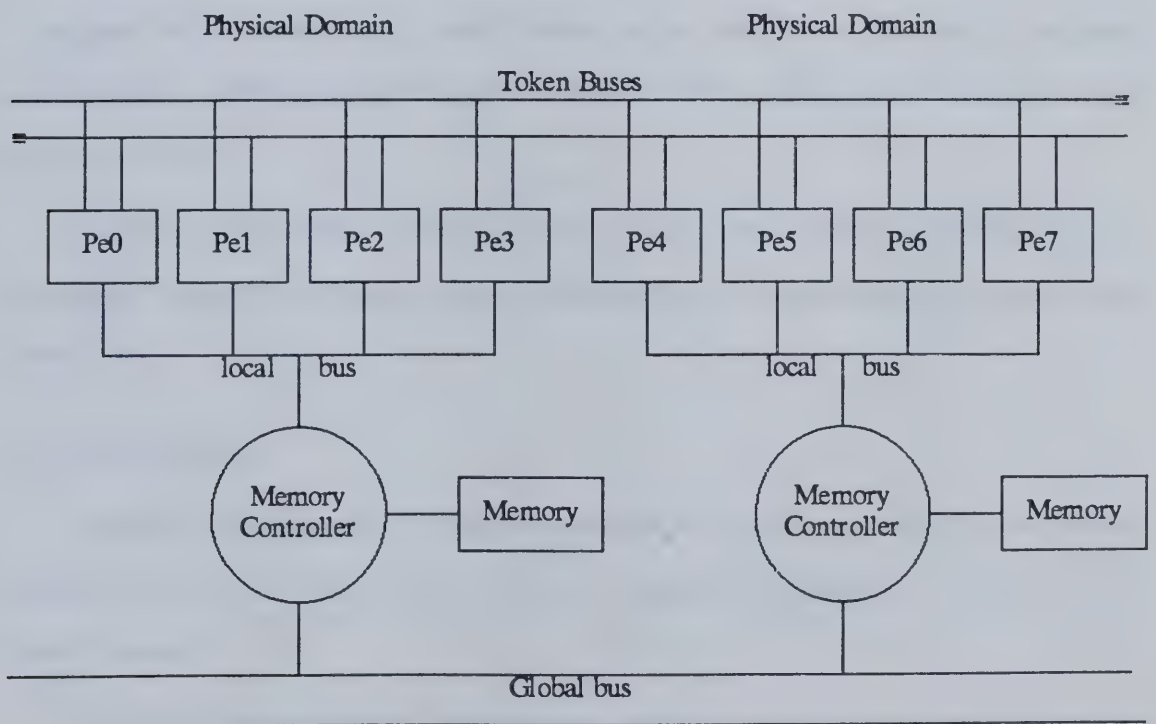



**Figure 2.4 Id Machine**

each token bus. The processing element examines the data token in the slot and if it has its address, the data token is removed from the slot and placed in an input queue to the sorter. The sorter matches this data token with others bound for the same instruction and if the required number of operands have arrived, the operands are packaged and sent to the code fetch unit. The required instruction is fetched from local memory and the stated operation is performed by the arithmetic-logical unit (ALU). An output unit takes results from the ALU and forms result tokens including the addresses of the destination processing elements. These result tokens are placed in slots on the token buses. All functional units within the processing element maintain first-in-first-out input and output queues.

A unique feature of this architecture is the unfolding interpreter which dynamically replicates program loops (iterations, procedures) to achieve more concurrency. More complex token identification (tags) and operators which alter tags are needed to support this interpretation. This identification scheme is used to assign activities to processing elements with locality in mind, (i.e. a block of program code is assigned to a physical domain) so that result tokens need not be transferred great distances. This machine also supports structured data types and accesses these by reference rather than by value.

The Id machine achieves concurrency through the many identical processing elements, independent functional units within each processing element, and through the unfolding interpreter.

### 2.4.4. Systeme LAU

Research on the Systeme LAU dataflow architecture is pursued at the CERT Laboratory in Toulouse, France. The architecture is based on a high level language LAU which is almost directly executed.

The machine [Pla76, SCH77] is composed of a processor subsystem (thirty-two identical processing units), a memory unit and a control unit (see Figure 2.6). The memory unit stores instruc-

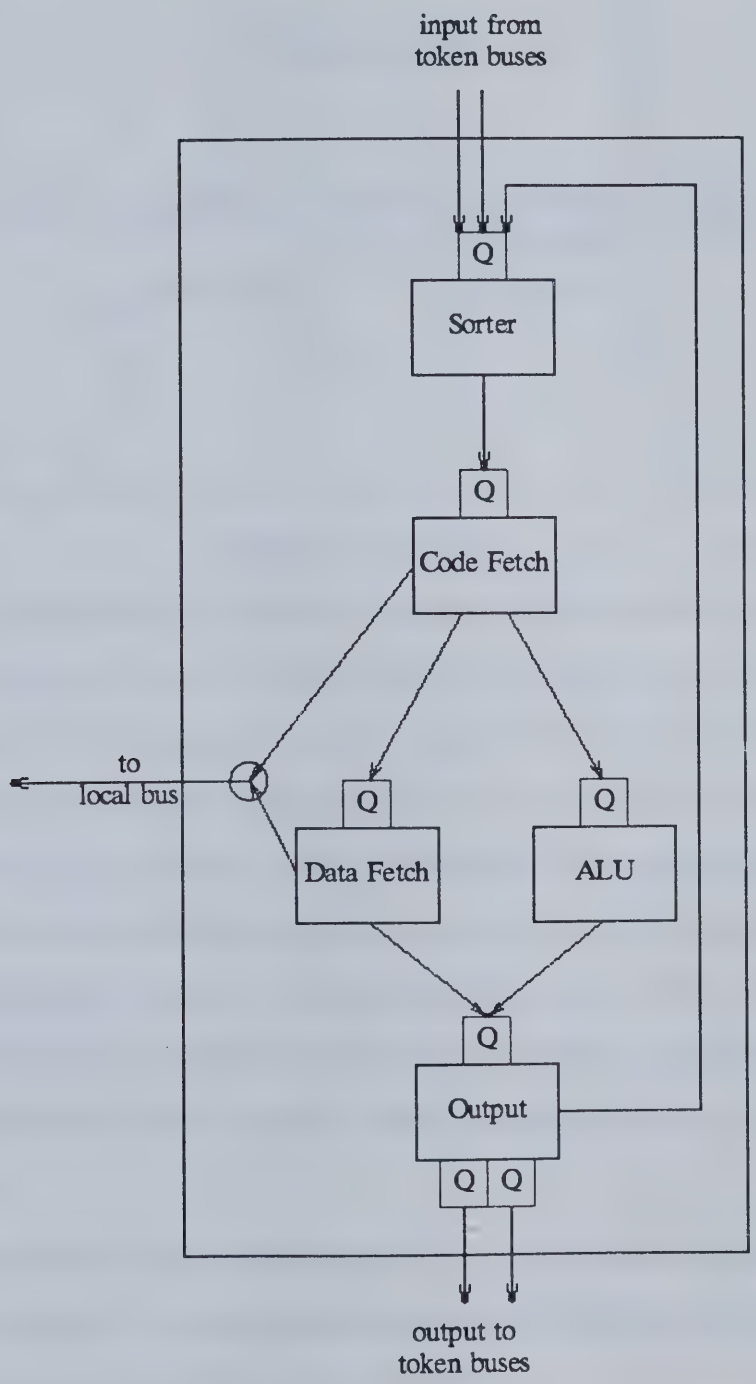



**Figure 2.5 Id Processing Element**

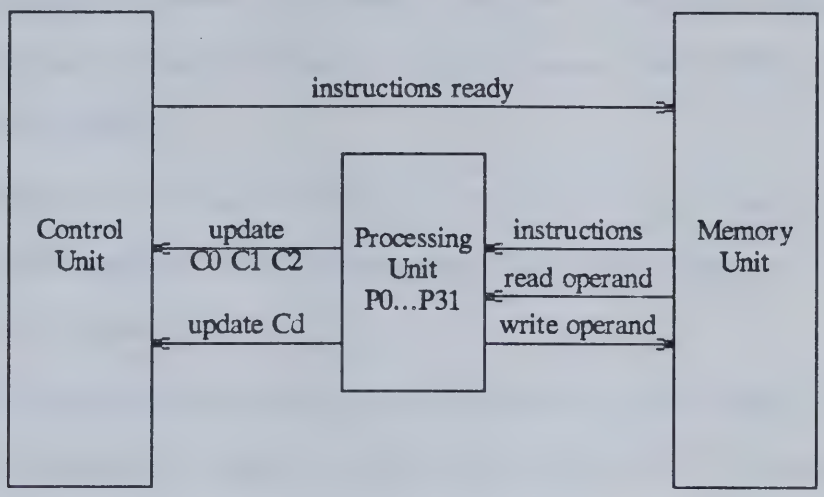



**Figure 2.6 LAU Machine**

tions and data. The control unit contains the Instruction Control Memory and Data Control Memory and uses these to determine which instructions are executable. The Instruction Control Memory is served by two functional units which update and search the Instruction Control Memory. The Instruction Control Memory contains three control bits for each instruction in memory and when these bits are set to all ones, indicating that all data operands are available, the address of the corresponding instruction is sent to the memory unit and the control bits are reset. The memory unit fetches the instruction and puts it on the input queue to the processor subsystem. An idle processor accesses this input queue and fetches the required operands and executes the instructions. Afterwards, results are written to memory and the appropriate bits in the control memory are set.

The unique feature of this architecture is the control unit which controls the enabling of instructions. Concurrency is supported through the thirty-two identical processing elements and the independent control, memory and processor subsystems.

### 2.4.5. Manchester University (MU) Machine

The Manchester University (MU) dataflow computer is described in Watson [WaG79]. Research has included the design of a high level dataflow language called LAPSE, which is directly supported by the architecture.

The machine is a ring composed of three units: matching store, instruction store and processing unit. A switch manages external communications, and a token queue acts as a buffer to the matching store (see Figure 2.7).

The matching store takes a token from the token queue and if this token is the only input required for an instruction, it passes it directly to the instruction store. If the token is one of a pair, the matching store searches for its partner and packages them together and sends them to the instruction store. If the partner is not present, the token is stored in the matching store. Upon receipt of a token packet the instruction store fetches the instruction and passes the packet to the processor unit. A distribution system within the processor unit sends the instruction packet to any idle processing element (a twenty processing element unit is under construction). The processing element performs arithmetic, comparison and token management operations. After execution the arbitration system within the processing unit produces one or two result tokens. The tokens proceed through a switch which also manages input and output with external sources and passes tokens on to the token queue.

All units operate asynchronously, thus parallelism is achieved through the independence of these units and through the identical processing elements within the processor unit.

Besides the value and destination field, tokens also specify the process, argument number, and iteration of the token. This tagging is similar to the Id machine and permits reentrant code. The token matching scheme is also similar to the Id machine but the matching and instruction stores are centralized rather than distributed.

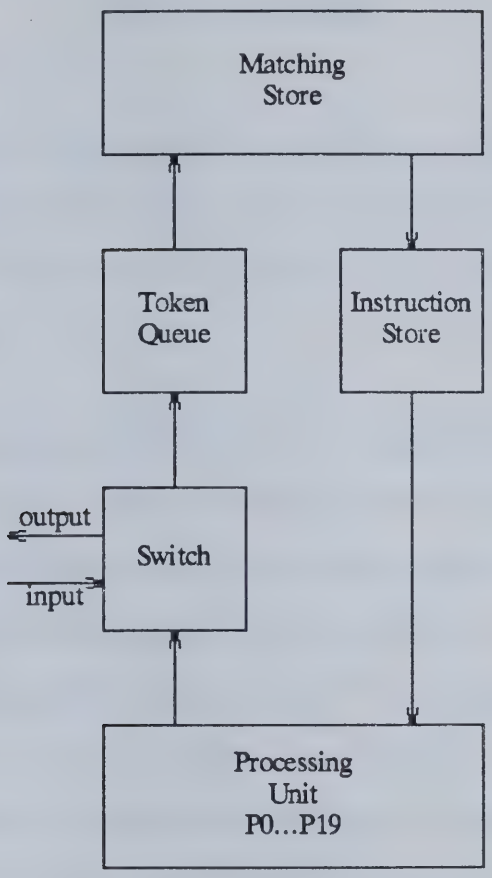



**Figure 2.7 MU Machine**

# CHAPTER 3

## Evaluation Methodology

Although the dataflow computing model is conceptually simple, the dataflow architecture proposals are often complex, involving separation and replication of function to an unusual extent. This complexity increases the need for evaluation during the early design stages.

### 3.1. Experimental Model

Presented here is an experimental design for the evaluation of dataflow computer architectures. In a more general sense the object is to study a proposed computer architecture with the intent of optimizing its predicted performance. To optimize performance, quantitative performance measures must be established and hypotheses made concerning which elements of the architecture (i.e. parameters) have a causal effect on those measures. With parameters and performance measures defined, experiments to study the architecture must be designed. The most direct way to study the architecture would be to implement the architecture as a physical machine and conduct tests by varying parameters. This approach is neither feasible nor desirable, because it would introduce another variable, the implementation. A more practical approach is the construction of a model whose behavior can be examined by simulating it on a computer.

A model is a representation of the behavior and structure of a system. A computer program uses this representation to generate behavior consistent with the model. This method of study is repeatable and many tests can be conducted in a short period of time. Models can be expressed in various ways: mathematical relationships, logical rules, formal languages, statistical probabilities.

The model which is proposed here (see Figure 3.1), describes the components of a computer architecture and their interaction using an architecture description language. The components of a computer architecture are the functional units and the data paths between them, while component interactions refer to the data exchange and means of controlling communication between

components. The architecture description is not at the gate or latch level but at the logical functional component level (i.e. endo-architecture).

The input variables of the model are a test computer program coded in the instruction set of the particular architecture, data required by that program and parameter values. The input variables (program code and data) will determine the behavior of the model, consequently this model

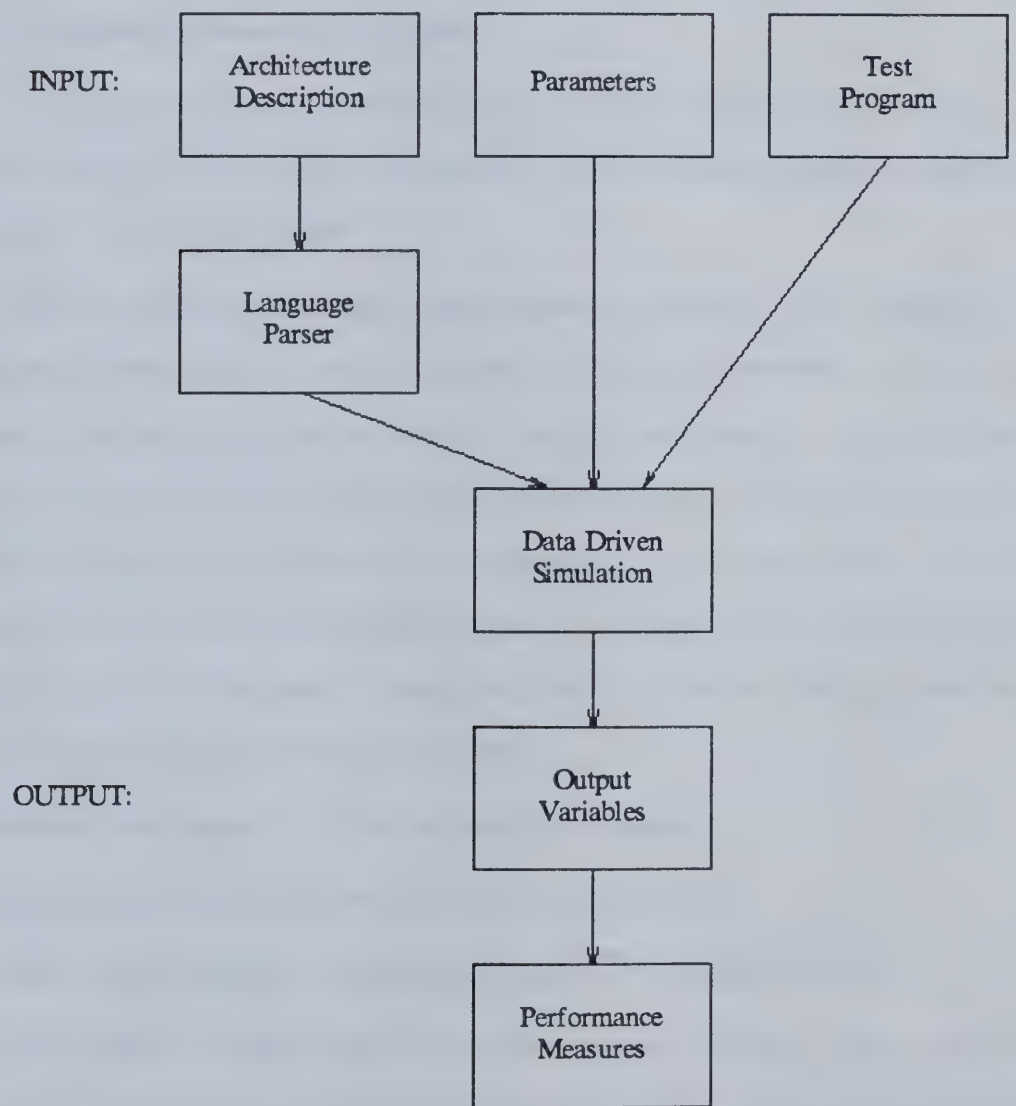



**Figure 3.1 The Modeling Process**

is data driven and deterministic, rather than stochastic.

A computer simulation program simulates the behavior of the computer architecture (i.e. it executes the program code). This is the simulation of the execution of a single job on a computer architecture not a multiprogramming job stream on a computer system. The simulation is monitored to produce statistics which summarize the behavior of the architecture and are used to produce performance measures.

### 3.2. Performance Measures and Parameters

The choice of performance measures and parameters for this study was based on both a review of previous research and an examination of the characteristics of dataflow architectures as discussed in the preceding chapter.

The United States Army/Navy Computer Family Architecture (CFA) Committee [FuB77] developed a methodology for quantifying relative performance of alternative architectures. They wanted to compare architectures independent of existing implementations. The comparison consisted of three phases: an initial ranking based on a set of absolute and quantitative criteria, a test program evaluation and consideration of software support and life-cycle costs. The first step reduced the set of nine candidate architectures to three acceptable architectures: IBM 370, PDP11 and Interdata 8/32. Those were evaluated using twelve test programs written in machine code. The following measures of performance were used:

S- number of bytes required for the test program in main memory

M- number of bytes transferred between main memory and the CPU.

R- number of bytes transferred among internal registers of the central processor.

The rationalization for these measures was that a higher S implied a higher cost for more hardware, a higher M or R implied a longer execution time. Implementation details (cache memory, instruction pipelining) can of course affect these conclusions but they are not characteristic of the architecture. Since a dataflow architecture achieves greater speed through simultaneous

execution of instructions, the CFA study assumption of sequential execution is violated. Therefore, M and R measures would not be indicative of execution time in a dataflow architecture.

It is important that any performance measures used to evaluate dataflow architectures consider concurrency. A standard von Neumann computer architecture does not specify instruction execution time, because this is normally considered part of the implementation. But it is impossible to evaluate dataflow architectures without estimates of relative time required by each functional unit (cycle time). It is necessary to incorporate the concept of time to determine which functions are being performed concurrently. Using these relative cycle times as parameters, the designer can determine an optimal set of cycle times, which can then be sought during the hardware design phase.

Ruby Lee [Lee80] of Stanford University describes an experiment which investigates the optimal number of processors in a parallel processing system. The measures of performance used by Lee were of four types: speed of execution of a test program, utilization of resources, compression of the computation and quality of processing. Speed of execution was measured by the parallel index (i.e. ratio of number of operations in the computation to execution time in steps) and by speed-up (i.e. ratio of execution time of the serial computation to that of the parallel computation). Utilization was measured by utilization (i.e. ratio of processor busy time to execution time) and efficiency (i.e. ratio of number of operations in the serial computation to the product of execution time and number of processors). Compression was measured by redundancy (i.e. ratio of number of operations in the parallel computation to number of operations in the serial computation) and compression (i.e. inverse of redundancy). Quality was a composite of other measures, defined as the product of speed-up, efficiency and compression. The parameters used by Lee were number of processors and configuration of the system.

Gostelow and Thomas [GoT80] report results of experiments conducted by simulating a proposed dataflow architecture. The performance measures used were execution time of a program, processor efficiency (busy time / execution time) and mean cycle time. The parameters were

number of processing elements, memory bus speed, problem size, different problems and assignment functions (i.e. mapping of operations to processing elements). With these performance measures and parameters it was possible to investigate optimal number of processing elements and bottlenecks in the system.

Results of simulation tests on the LAU dataflow architecture are reported by Plas [Pla76]. The performance measures used were execution time and average parallelism achieved. The parameters used were number of functional units (processors, memory units), cycle time of functional units, parallelism of the problem and varying source programs.

The choice of performance measures and parameters is of course dependent on the purpose of the investigation. The present purpose is to help make decisions regarding the endo-architecture of a proposed dataflow architecture. By re-examining the characteristics and architecture attributes presented in Chapter 2, it can be seen that the object is to help answer questions about the structure, behavior and program representation of the architecture. These are rather arbitrary divisions since all elements of the endo-architecture are interdependent. Usually the designer makes decisions about the program representation first and then attempts to find an appropriate structure. Experimental investigation is required first at the structural and behavioral levels, where relevant questions are:

(1) What functional units are required? A decision on the separation of function implicitly defines the execution cycle. The distribution of program and data storage determines the nature of memory (central versus distributed).

(2) How many of each functional unit are required? The replication of function determines how parallelism is achieved through the structure. Are identical or specialized functional and computing elements desired?

(3) What communication paths will connect the units? Are control paths included as well as data paths?

(4) How will dynamic mapping of instructions to computing elements and static mapping of programs to storage be determined?

(5) What will be the relative cycle times of functional units and speed of buses?

(6) What problems are best suited to the architecture?

(7) Is there a specific amount of concurrency which best fits the architecture?

Performance measures and parameters which are appropriate to both the class of dataflow architectures and the goals of this thesis are listed below.

Performance Measures

(1) execution time (ET)

(2) utilization (U)

- of one functional unit (U=busy time/ET)
- of system (U=total busy time/n*ET
  where n=number of functional units)

(3) speed-up (ET sequential/ET parallel)

(4) efficiency (ET sequential/n*ET parallel)

(5) average wait time (the average time a token waits for a functional unit)

Parameters

(1) configuration: what and how many functional units

(2) relative cycle times of functional units

(3) test programs

(4) degree of concurrency of the problem

### 3.3. Architecture Description

An important decision to make in the modeling process is how to represent the computer architecture. The choice of a representation depends on both the purpose of the modeling process

and the level being investigated. The purpose here is to represent an evolving design within the class of dataflow computer architectures, where modeling results will indicate possible improvements to the design. These alterations will then be incorporated into the current design for reexamination. Given this purpose, what characteristics are desired in a representation?

The representation must be suitable for all proposals within the class of dataflow architectures, therefore a simulator developed for one particular architecture which implicitly describes the architecture through its behavior is not appropriate. The representation must be explicit, objective and suitable to the whole class of dataflow architectures.

In order for a description to be easily changed it must be readable by humans. Humans find it easier to understand descriptions which are represented familiarly, are modular in structure and are presented in more than one manner (e.g. visually using diagrams and logically using notation). Various points of view and levels of abstraction (e.g. overall structure and the detailed behavior of one unit) also contribute to understanding.

Precision is another required characteristic since an optimizing process hopes to identify precisely the factors affecting performance.

As the optimizing phase is only one phase in an overall design process, it is reasonable to expect the description in this phase to contribute to the construction of descriptions at later stages. This description should translate into more detailed designs at the hardware level and therefore should be understood by hardware designers as well as computer architects. Also the representation should be machine readable so that translations and changes can be performed by machine thus contributing to an automated design process.

Computer architectures are often informally described through block diagrams showing structure and interconnections (as in Chapter 2) with accompanying natural language descriptions. Although this method is appropriate for overall understanding of general principles, it is not precise and fails for complex designs. A more formal representation would use mathematical or logical notation or a programming language.

Especially appealing is a computer programming language since computing professionals are already familiar with this form of representation. The block structured programming languages encourage modularity and allow alternate levels of abstraction. Computer programming languages are machine readable thus allowing easy translation to other design levels. Creating or changing a description using a programming language is facilitated by many tools such as compilers and editors, which promote automated development. General purpose programming languages have been used to describe computer architectures, but a better mapping between architectural components and language constructs can be achieved using an architecture description language (alternately called computer hardware description language or computer design and description language). An architecture description language is an appropriate representation for an architecture description in an optimizing design process.

Since the endo-architecture is to be examined, it must be determined how this level affects the choice of architecture description language. Within a single user dataflow computer system or subsystem, under examination are the functional units, communication paths and execution cycle behavior. The basic entities [Mak82] of such a system are: instruction set, data tokens, control tokens, and a set of cooperating processes. A dataflow instruction set will contain ordinary arithmetic, relational and logical operators, control constructs to control execution paths and data structure manipulation operators. Data and control tokens may be of variable length, accessed by bit rather than by word.

A dataflow architecture is fundamentally a set of cooperating processes, executing concurrently, initiated asynchronously by the flow of data tokens. A dataflow process abstracts easily to the concept of an asynchronous module as discussed by Keller [Kel74]. An asynchronous module is independent and its "activity is regulated by initiation and completion signals, with no clocks being present" [Kel74, p. 21]. This abstraction to a set of asynchronous modules exists at many levels, but the concern here is with the system architecture level where a processing element (e.g. ALU, sorter, memory controller) is activated upon receipt of a data token. The result of this

activation is a new data token sent to another processing element. The results of computation are not affected by the order of arrival of tokens. Theoretically, there are no side effects and all global variables are represented as data abstractions (i.e. protected within a monitor).

Processes are likely distributed throughout the system on independent physical components. Token passing may therefore require external communication paths such as buses. Since processes operate concurrently, and independently send messages to other processes, facilities must exist for queuing messages upon arrival. A dataflow architecture is often a collection of identical processors or subsystems, and processes may contain complex data objects such as associative memories, queues and stacks.

In summary, a description of a dataflow architecture at the endo-architectural level must describe instruction set, tokens and processes executing concurrently. Particular properties of these have been cited above. Can existing architecture description languages accommodate such a description?

Many architecture description languages have been developed. A comparison of computer design and description languages is presented by Makarenko [Mak82] and a general discussion of description language research is presented by Dasgupta [Das82]. ISPS [Bar79] is the best known hardware description language and was developed from ISP, first proposed in 1971 [BeN71]. It is a general purpose language and forms the base of a computer-aided design facility at Carnegie-Mellon University. ADL [Leu79] is a specialized language designed to describe packet communication systems which are systems composed of modules communicating by sending information packets to each other. An ADL description is organized into modules, each with a specified input and output and further division into sub-modules which contain behavioral descriptions. Sharing of data is controlled by monitors within a module. Sub-modules are easily replicated and the number of replications can be based on a parameter data type. Since a dataflow architecture is a packet communication system, ADL would have been an acceptable choice for this study.

The architecture description language used in this study is $S_A^*$, a general purpose language developed as part of a family of description languages [Das81, Das82, Das83]. $S_A^*$ was chosen because an $S_A^*$ simulation facility exists at this university, thus providing an opportunity to test and extend the use of these tools to the class of dataflow computer architectures. $S_A^*$ is a block structured language in the likeness of Pascal, whose highest level modularization construct is the system. A system may be composed of other systems but at the lowest level it is composed of mechanisms. The mechanism is the most important structuring construct. It acts as a critical section and as a data abstraction block, such that variables declared within the mechanism are accessible only to procedures within that mechanism. Mechanisms are initiated through calls to their public procedures; the calls are queued if the mechanism is already active. Although $S_A^*$ is basically procedural, mechanisms can be initiated at start up and can continue execution without waiting for the completion of other mechanisms which they have activated. Consequently, mechanisms will execute concurrently. The primitive data type is the bit, but user defined types are allowed and can be composed of the following data structures: sequence, array, tuple, stack and associative array. As well as the usual Algol-like constructs, $S_A^*$ includes an operator which denotes concurrent execution of statements. Although synchronization constructs are available, there is no way to refer to absolute time or to associate a delay time with any construct. The mapping of $S_A^*$ constructs to dataflow components is discussed in Chapter 4.

### 3.4. The Simulation Facility

The $S_A^*$ simulation facility was developed by Makarenko in 1982 [Mak82] and has not previously been used as a research tool. Although the simulator does not implement all $S_A^*$ constructs, it does provide an adequate subset. The simulator produces the following statistics: the number of times a sequence variable is read or written, and the number of times a mechanism is activated. As a first step in the simulation process, an $S_A^*$ description is input to the compiler which produces a parse tree. The parse tree is used as input to the simulator, whose execution is controlled by user commands. Thus the user can assign values to variables within the description and introduce

the code of a test program. During a simulation, concurrent execution of mechanisms is achieved by executing each active mechanism in turn for one time step. The scheduler simply advances cyclicly through the list of mechanisms, and whenever an active mechanism is encountered, the first executable statement on that mechanism's executable statement list is executed. In this manner, an appearance of concurrency is achieved. All calls to a procedure whose encompassing mechanism is already active are queued in incoming order.

The simulator described above did not provide statistics which could be used to calculate the measures specified previously as desirable for the evaluation of dataflow architectures. The statistics required were: test program execution time, system and functional unit busy time and average wait time for each functional unit. Also, the simulator implementation of the $S_A^*$ activate construct was not appropriate to dataflow asynchronous processes. Extensions and modifications to the simulation facility were undertaken to incorporate the above requirements. The changes involved new input and data collection within the simulator, a new implementation of the activate construct, and a new scheduling algorithm. As well, the simulator command language was extended to generate a report. The modifications made to the simulator are described in the remainder of this section.

All of the performance measures mentioned above require the concept of time and require the maintenance of cumulative elapsed time of each parallel execution path during the simulation. Time can be introduced by associating an execution time with each statement, procedure or mechanism. Because a mechanism contains procedures, it is too gross for accurate timing. Associating execution time with each statement would provide more detail than is needed at the endo-architecture level and would require information too specific for this phase of the design process. The procedure appears to be the right level for timing because it provides a suitably flexible block which the designer can subdivide if more detail is desired.

The next consideration is how to incorporate time into this facility. Should the language itself be extended to associate a delay time with each procedure declaration or should the simulator

be modified to accept procedure execution times? Since relative cycle times are to be used as parameters in the evaluation, it is reasonable that procedure times should be an input to the simulator. Changing the times for tests will then be convenient since they will be together in a separate input file rather than distributed throughout the $S_A^*$ description. Therefore, it was decided to associate a fixed execution time with each procedure and input this data as a separate file to the simulator.

In order to collect execution time, busy time and wait time statistics, it was necessary to add data structures and variables within the simulator. First an extra field was added to the procedure record to store procedure execution time. Next the mechanism record was extended to store the mechanism total busy time, total wait time, real time and earliest pending call (of current queued calls). Every time the mechanism is activated (i.e. one of its procedures is invoked) the total busy time is increased by the execution time of the called procedure. Total wait time is also updated at this point to include the length of time the procedure call waited for this mechanism. Real time is calculated by selecting the larger of the mechanism's current real time and the time this procedure was called, and adding the procedure time. The real time is therefore the cumulative elapsed time of that execution path. The earliest pending call field is updated only when a new call is queued for that mechanism or when a call is removed. The queued calls are a list of pending calls sorted by time called, except that private procedure calls are always put on the front of the list. The execution of a *call* or *activate* statement inserts the real time of that mechanism as the time called field on the pending call record attached to the called mechanism. When execution of a test program is complete and simulation stops, statistics can be collected for reporting. Each mechanism record is examined to determine the maximum real time, for this is the longest time required by any execution path and hence is the execution time of the test program. Mechanism busy time is directly available and average wait time is calculated by dividing total wait time by number of activations.

A new method of scheduling mechanisms was required since the previously used round robin method would execute mechanisms in the wrong order and invalidate the real time of mechanisms.

For example, if two mechanisms each queued a call at five and ten time units respectively, then the call at time five must be executed first because during its execution it could call the other procedure and this call should be put on that queue before the previous call at time ten. The new scheduling algorithm maintains the list of mechanisms sorted in ascending order of earliest pending call, those mechanisms with no pending calls are on the end of the list. The list is updated when a procedure terminates and also when a new call is queued for any mechanism. Mechanisms are selected from the front of the list for execution.

Another alteration to the internal operation of the simulator involves the implementation of the activate construct. The call construct was implemented such that the caller became inactive and waited for completion of the called routine which might return results through parameters. In contrast, the activate construct suspended the caller until allocation (passing arguments and activating the mechanism) of the called routine, at which time the caller became active again and resumed execution. No results could be passed back at completion of the called routine. In both cases, the caller was required to wait for allocation of the called routine. This implementation was not compatible with the asynchronous initiation and concurrency of processes in the dataflow model, which requires that the caller not wait for any length of time. Consequently, the *activate* implementation was changed so that the caller does not wait for allocation. Instead, execution of the *activate* statement creates a new record to hold the values of the arguments passed. This argument list record is attached to the queued call and execution of the caller continues. In effect, the actual parameter is now bound to the formal parameter at the time of the call rather than at invocation. Makarenko chose to bind parameters at the time of invocation because

> There are very few hardware components that operate in a fashion where they can be activated from a number of different spots, with no acknowledgement given to the caller. [Mak82,p. 91]

However, this form of activation is precisely that of data flow processes, consequently binding must be done at the time of the call so that the caller can be released to continue execution. Otherwise, mechanisms would not be executed in the correct order.

Given this implementation change and the above method of calculating elapsed time, restrictions must be applied to the use of *activate* and *call* in the $S_A^*$ descriptions of dataflow architectures. All calls on public procedures must be *activate* statements, so that concurrent execution of the two mechanisms occurs and to insure the one way communication (message passing) demanded by dataflow. The *call* statement can be used only to call private procedures (i.e. local to the mechanism). The constructs, signal and await, which allow synchronization of mechanisms, cannot be used since the calculation of elapsed time would then be incorrect.

In addition to the above changes, the command language of the simulator was expanded in order to request and calculate the required statistics. The new command will generate a report summarizing the simulation test.

# CHAPTER 4

## Experimental Validation

### 4.1. Examples Using the Proposed Methodology

In the preceding chapter a methodology for evaluating dataflow computer architectures has been described. This chapter shows its application to two dataflow computer architecture designs taken from the current research literature: the Id (Irvine Dataflow) machine and the MIT machine. Standard configurations of the machines have been defined and then described using $S^*_A$. A test program was chosen and coded in the machine language of each machine. This program code was executed on the simulated standard machine and on variations of the standard configuration. Performance statistics were collected and the results are presented and discussed in Section 4.2.

Many papers have been published discussing the Id and MIT machines. Ongoing research has produced variations, extensions and improvements to the original designs, therefore there is no definitive MIT or Id machine. Through examination of published papers, reading between the lines and making reasonable assumptions where information is incomplete or not precise, an *MIT design* and an *Id design* have been assembled to serve the purpose of this experiment. A standard configuration of each design has been defined. This configuration incorporates minimum concurrency and thus provides a basis for comparison of other configurations which do incorporate concurrency.

The results presented in this chapter cannot be used to compare the performance of the two machines. Published papers are by necessity usually brief and therefore details of designs are not complete or precise enough to make direct comparisons between two designs. Rather, the results demonstrate how the proposed methodology can be used to investigate characteristics of a design.

### 4.1.1. Construction of an $S_A^*$ Description

Each independent functional unit within an architectural design is mapped to a *mechanism* in $S_A^*$. The mechanism construct encapsulates a critical section, as described in Chapter 3. Only one invocation of a procedure within a mechanism is active at any one time but many different mechanisms can be active simultaneously. In these dataflow designs, a mechanism is activated by an *activate* statement within some other mechanism. All parameters are evaluated at the time of the *activate* statement and the call queued if the mechanism is already busy. Parameters are not passed back at completion. Hence functional units are activated asynchronously and communicate with each other only by passing tokens (parameters).

Execution times are associated with procedures within the mechanisms, hence the experimenter can determine the timing detail by the number of procedures used. Global variables are used only when a variable is accessed by more than one mechanism. For the convenience of this experiment, all variables have the same size (one word of thirty-two bits).

### 4.1.2. Id Design

The Id machine is briefly described in Section 2.4.3 (see Figures 2.4 and 2.5). The standard configuration of the system used for simulation purposes is based on Gostelow and Thomas [GoT80] and consists of one local memory, two token buses and one processing element composed of

(1) sorter unit - matches data tokens and determines if an activity can be enabled

(2) code fetch unit - requests instruction code from memory

(3) ALU unit - executes instructions

(4) output unit - creates data tokens and communicates with token buses.

The only concurrency possible in this standard configuration is through pipelining of the instruction cycle, within the one processing element. Additional concurrency can be introduced by replicating processing elements. The $S_A^*$ description of this configuration is shown in Appendix A.

The instruction set operators are as defined by Arvind [AGP78]. An instruction code format (see Table 4.1) and data token format have been defined based on information in the same report. A data token is defined as the composition of the following fields:

< processing element address, destination activity name, value, port, number of inputs>

where the activity name is composed of

<context, code block, statement, iteration>.

Functional unit execution times are as stated in Gostelow and Thomas [GoT80] and are shown together with the mapping of functional units to $S_A^*$ mechanisms in Table 4.2. The

**Table 4.1 Id Instruction Format**

| Id Instruction Format | |
|---|---|
| Field | Contents |
| 1 | statement number |
| 2 | opcode |
| 3 | constant operand (0 or 1) |
| 4 | constant port (0 or 1) |
| 5 | constant value<br>or number of true destinations (opcode=switch)<br>or code block value (opcode=L) |
| 6 | number of destinations |
| 7 | destination statement number |
| 8 | destination port |
| 9 | number of inputs required by destination |
| | repeat fields 7,8,9 as required for each destination |

**Table 4.2 Id Functional Unit Execution Times**

| Id Functional Unit Execution Times | | |
|---|---|---|
| Id Unit | $S_A^*$ Mechanism | Execution Time (time steps) |
| Token Buses | ring | 4 |
| Memory | mem read | 6 |
| Sorter | sort | 4 |
| Code Fetch | fetch<br>receive | 1<br>1 |
| ALU | alu | 10 |
| Output | output<br>ring interface | 4<br>0 |

execution times used here may be unrealistic but the numbers themselves are irrelevant since the purpose is to demonstrate the method, not to perform an actual evaluation.

No data structure operations are supported in this standard configuration.

### 4.1.3. MIT Design

The MIT machine is briefly described in Section 2.4.1 (see Figure 2.2). A standard configuration has been defined based primarily on Dennis and Misunas [DeM75], including conditionals and iterations but excluding the multi-level memory system. Acknowledgement signals as described in Dennis [Den80] are included. The standard configuration used to obtain simulation results is as follows:

(1) one memory unit - not separated into instruction cell blocks, therefore no concurrency is possible during the enabling of instructions

(2) one arbitration network

(3) one operation unit capable of performing all arithmetical, logical and control distribution operations

(4) one distribution network

(5) one control network.

The $S_A^*$ description of the standard MIT machine is shown in Appendix B. The standard configuration incorporates no concurrency except pipelining of the instruction cycle. Concurrency can be achieved by allowing each instruction cell in memory equal access to the switching networks, and through the separation of the operation unit into one unit for each opcode.

Assignment of execution times to functional units is arbitrary since relevant information has not been published, but the assignment of times is such that the cycle time of the MIT machine is equal to that of the Id machine. Cycle time refers to the minimum time required for a token to complete one path through the system. Functional unit execution times are shown in Table 4.3.

**Table 4.3 MIT Functional Unit Execution Times**

| MIT Functional Unit Execution Times | | |
|---|---|---|
| MIT Unit | $S^*_A$ Mechanism | Execution Time (time steps) |
| Memory | memory | 6 |
| Arbitration Network | arbit net | 6 |
| Operation Unit | operation | 10 |
| Distribution Network | dist net | 4 |
| Control Network | control net | 4 |

A data token is composed of the following fields

<destination instruction, value, producing instruction>.

A control token includes an extra tag field to indicate whether it is to be used as a flag or as a value. Operation packets and instruction formats are as described in Dennis and Misunas [DeM75] with the addition of fields to implement acknowledgement signals.

The MIT machine was chosen as the second experimental machine[1] because it differs in many characteristics from the Id machine, thereby demonstrating the versatility of the proposed methodology. The MIT machine differs from the Id machine in the following respects:

(1) The MIT machine has fixed size instructions with a fixed number of operands and destinations. This necessitates extra operators (link operators) to replicate data tokens.

(2) Token storage rather than token matching is used. Operands are stored with instructions in memory, therefore different instantiations of the same instruction cannot be active simultaneously.

(3) Acknowledgement signals are used rather than unique activity names, therefore each instruction must acknowledge each operand to its producer.

---

[1]The Id design was chosen as the first example for this research because a simulation study has been reported in the literature [GoT80] and also because much information has been published about the design itself. The second design was chosen as a contrast to the first. Both the Manchester and Utah dataflow architecture proposals were rejected because they employ behavior similar to the Id design. The LAU design was not considered because not enough detailed information is available about the design.

(4) Concurrency is achieved by allowing each instruction cell in memory equal access to the switching networks, and through the separation of the operation unit into one unit for each opcode. Concurrency in the Id machine is achieved by replicating the whole processing element, not by replication and separation of individual functional units within the whole.

(5) Since there is only one memory and only one system there is no need to assign instructions to processing elements, as in the Id machine.

### 4.1.4. Test Program

The test program used to obtain experimental results integrates a function f, over the interval (a,b) using n intervals of size h, by summation of trapezoids. A high level language representation is shown below.

```
sum ← (f(a) + f(b)) / 2
x ← a + h
for i ← 1 to (n - 1) do
        y ← f(x)
        sum ← sum + y
        x ← x + h
endfor
sum ← sum * h
```

The Id machine language program [AKP80] is shown in Figure 4.1. The operators which are not self explanatory are L, $L^{-1}$, D, and $D^{-1}$. These operators change portions of the activity name, which is composed of a context, code block, statement number and iteration number. The code block and statement number are statically assigned and the other two are dynamically altered during execution. The L or loop operator creates a new and unique context by changing that part of the activity name. The $L^{-1}$ operator returns the context to its previous value. The D and $D^{-1}$ operators adjust the iteration number. The switch operator merely routes data tokens depending on the true or false value sent by a decision operator.

The MIT machine language program is shown in Figure 4.2. The large black circles represent input values. The small black nodes represent the *link* operator which replicates a token

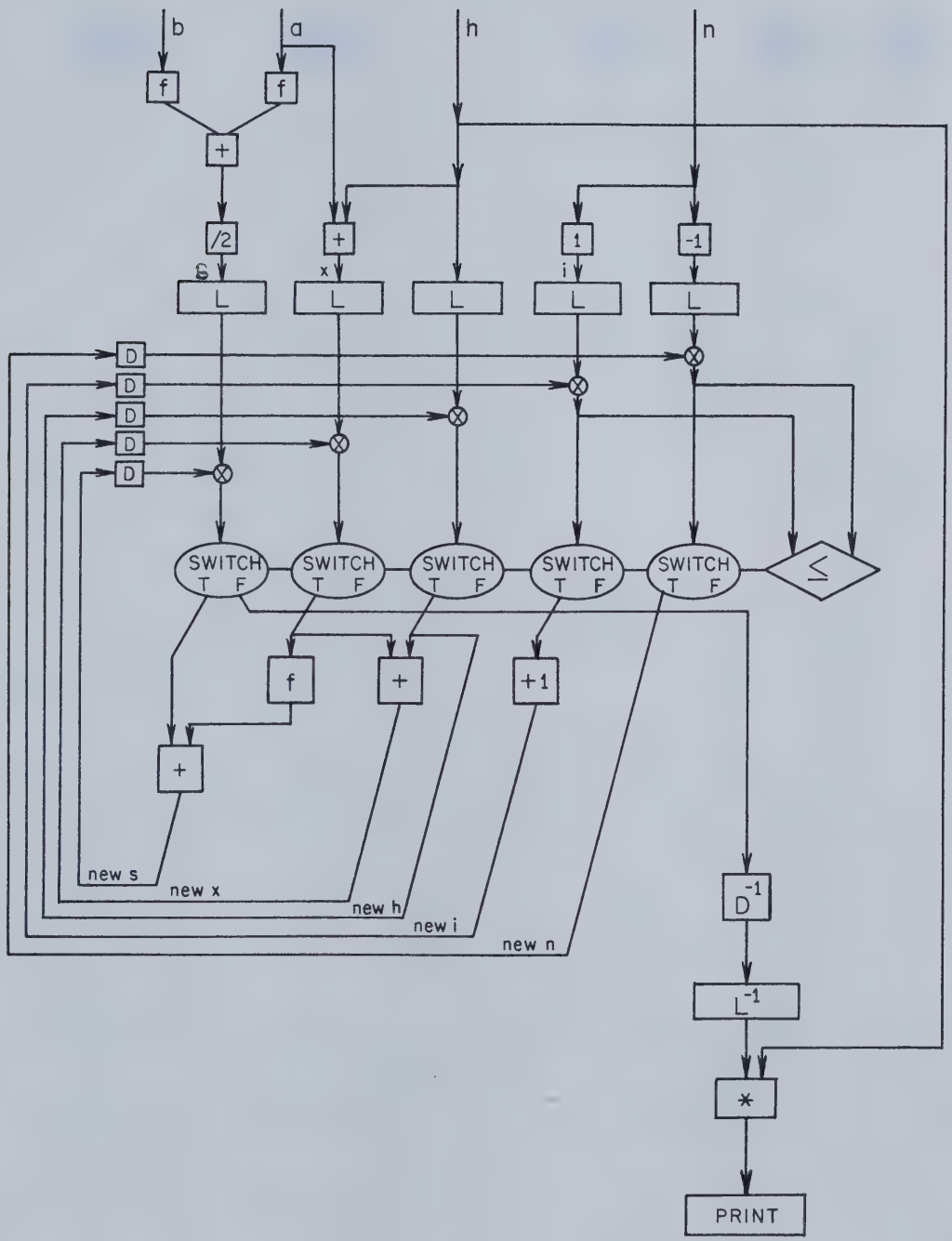



Figure 4.1 Id Machine Language Program: Integration

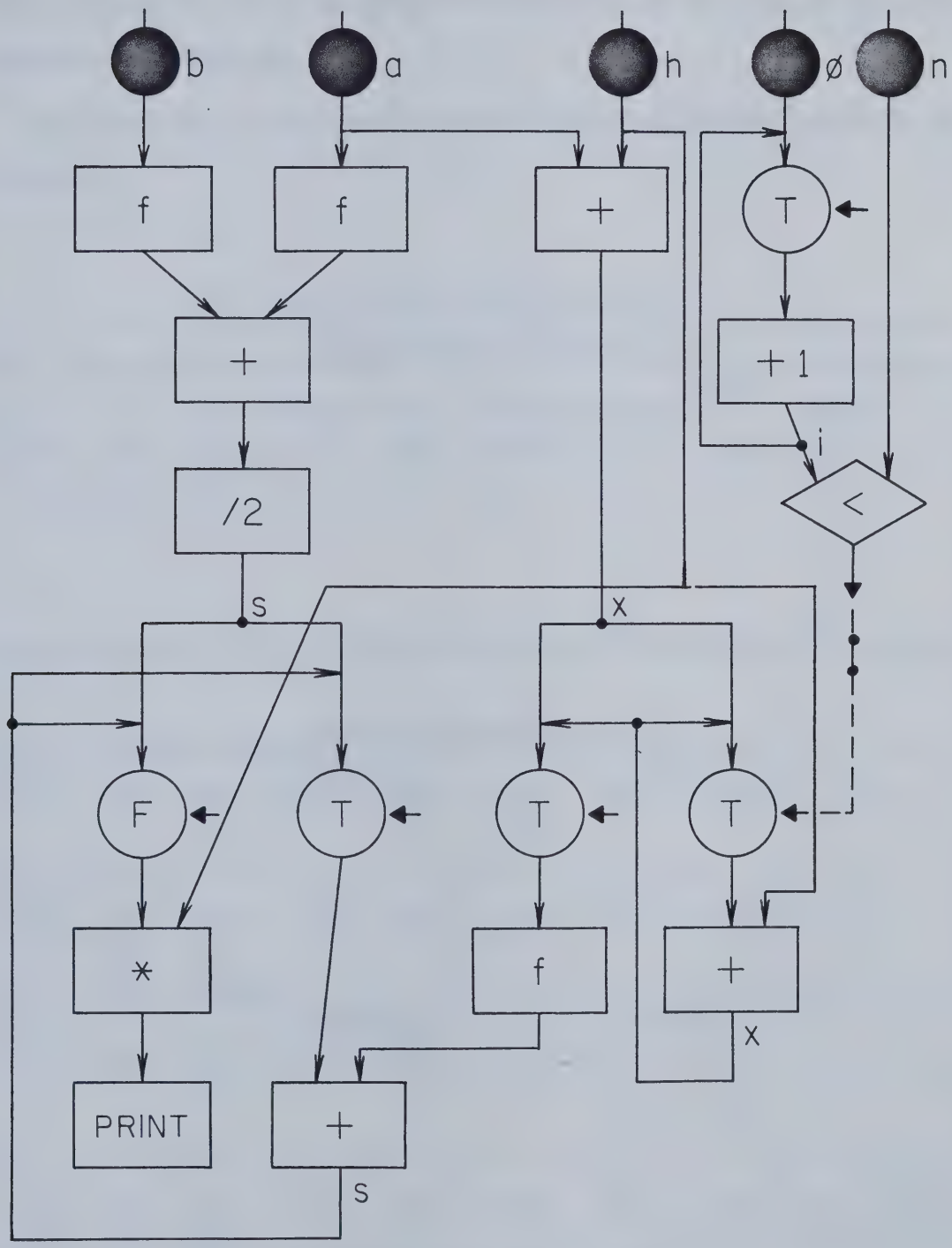



Figure 4.2 MIT Machine Language Program: Integration

and sends it to two other operators. The large circles (either True or False) are gates which permit the passage of a token depending on the value of a gate flag received as a control token from a decider operator. In summary, the machine language instructions are represented as rectangles, diamonds or small black nodes.

Examples of actual machine code instructions for both the Id and MIT machines are shown in Table 4.4.

**Table 4.4 Examples of Machine Code Instructions**

Example of Id Machine Code Instructions

| statement number | opcode | constant | constant port | constant value | number of destinations | destination statement number | destination port | destination number of inputs |
|---|---|---|---|---|---|---|---|---|
| 4 | divide | yes | 1 | 2 | 1 | 8 | 0 | 1 |
| 23 | <= | no | - | - | 5 | 18 | 1 | 2 |
| | | | | | | 19 | 1 | 2 |
| | | | | | | 20 | 1 | 2 |
| | | | | | | 21 | 1 | 2 |
| | | | | | | 22 | 1 | 2 |

Example of MIT Machine Code Instructions

| statement number | type | format | acknowledgements received | acknowledgements required | opcode | destination tag | destination statement |
|---|---|---|---|---|---|---|---|
| 63 | I | 01 | 1 | 1 | add | - | 32 |

| statement number | type | gate value | gate flag | gate ack. | data value | data flag | data ack. |
|---|---|---|---|---|---|---|---|
| 64 | D | true | - | - | - | - | - |
| 65 | D | constant | - | - | 1 | - | 1 |

| statement number | type | format | acknowledgements received | acknowledgements required | opcode | destination tag | destination statement |
|---|---|---|---|---|---|---|---|
| 33 | I | 06 | 3 | 3 | link | gate | 16 |
| 34 | I | - | - | - | - | gate | 64 |

| statement number | type | gate value | gate flag | gate ack. | control value | control flag | control ack. | destination 3 tag | destination 3 statement |
|---|---|---|---|---|---|---|---|---|---|
| 35 | D | no | - | - | - | - | - | value | 38 |

The iteration loop in this program demonstrates a fundamental difference between the Id machine and MIT machine. Because Id employs token matching rather than operand storage within instructions, more than one copy of an instruction can be active at one time. Therefore instructions from different loop iterations can execute simultaneously. This is not possible in the MIT machine since operands are stored with the instruction and a producer instruction must receive acknowledgement of its result before it can be enabled again (only one copy of an instruction is active at a time).

Within the *for* loop three operations can be performed simultaneously:

$$(y \leftarrow f(x)),\ (x \leftarrow x+h),\ \text{and}\ (i \leftarrow i+1).$$

The Id machine language program can also simultaneously compute (sum ← sum + y) from the previous *for* loop and prepare new data tokens for the variables h and n, thus allowing six simultaneous operation executions.

## 4.2. Results

### 4.2.1. Id Design

The standard Id machine is used as a basis for the comparison of three other system configurations: system #2, #3 and #4, consisting respectively of two, three and four processing elements.[2] Figure 4.3 shows execution time of the test program for the four systems. Systems #2 and #3 both executed the test program in less time than the standard machine, with system #2 achieving the best performance. In the standard machine all output tokens are transferred directly to the input queue of the sorter unit, thus bypassing the token buses. Therefore, the token buses carry tokens only during the start up phase of execution when input tokens are present. As the number of processing elements within the system increases, the probability that a token is consumed by the

[2]A system including more than one processing element must assign each activity (enabled instruction) to a particular processing element for execution. This is done using a simple assignment function based on statement number and context. An activity with context $u$ is assigned to physical domain $x$ where $x$ = ( $u$ **mod** number of physical domains). The systems in this experiment include only one physical domain. Within a physical domain, statement number $s$ is assigned to processing element $pe$ where $pe$ = ( $s$ **mod** number of processing elements).

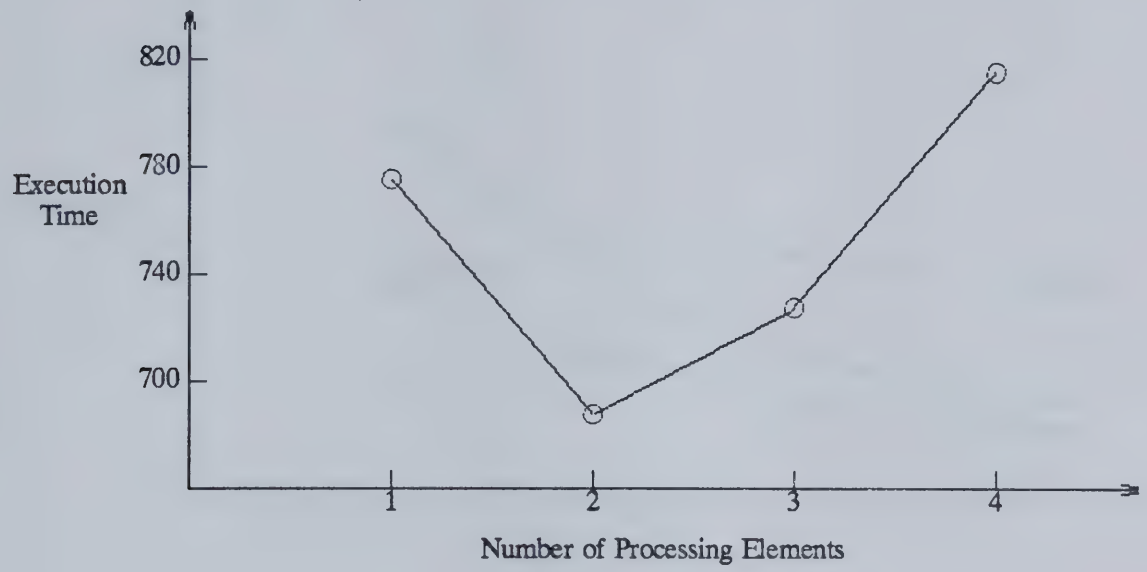



**Figure 4.3 Id Execution Time**

processing element which produced it decreases, hence more tokens must travel on the token buses. Concurrent execution thus exacts the price of time spent on the token buses. The execution time of system #4 shows how token traveling time can outweigh all time gained by concurrent execution.

Utilization (busy time/execution time) by system configuration is shown in Figure 4.4. Three points are shown for each system configuration: overall system utilization, ALU utilization and memory utilization. The ALU utilization shown is the maximum attained by any one ALU unit. The ALU utilization of the standard system is 86%, indicating that start up and data dependencies prohibited ALU operation only 14% of the time. The decreased ALU utilization of the other systems reflects the decreased execution time of an individual ALU and the increased communication time of tokens on the token buses.

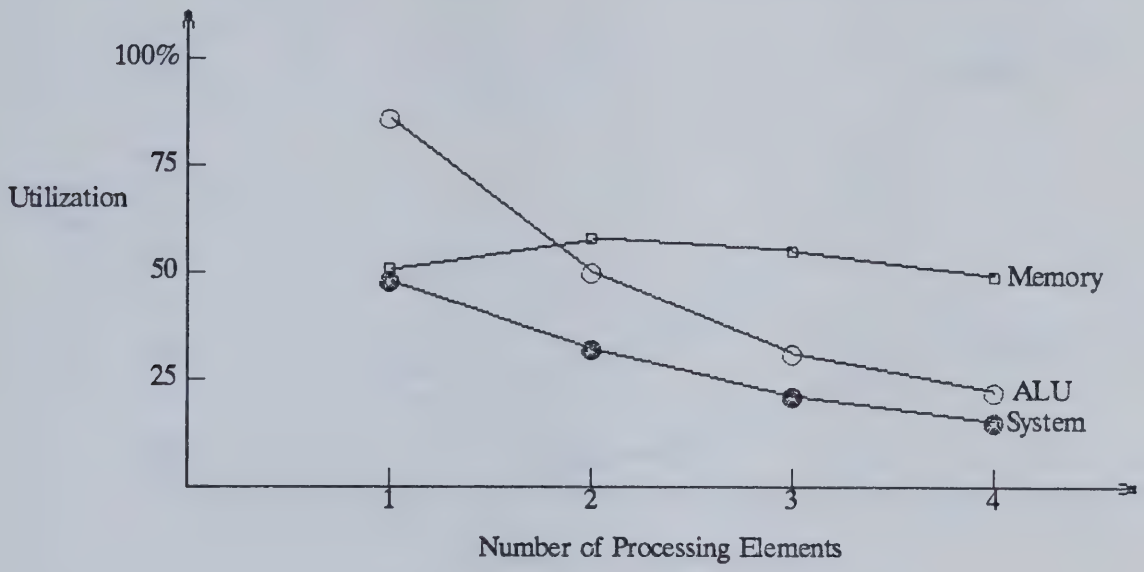



**Figure 4.4 Id Utilization**

Remember that the functional unit execution times used here are reasonable but arbitrary. Functional unit execution time can be used as a parameter to investigate workload balance between functional units within the system. By changing one functional unit execution time (token bus execution time was doubled), the communication costs are exaggerated. Figure 4.5 shows how bus time is critical to the speed of the system. There are no gains in performance and in fact execution time increases with the addition of processing elements. Functional unit execution time is an important parameter in the evaluation of workload balance.

No experiments were conducted with a system containing more than four processing elements. The Id design groups each set of four processing elements into a physical domain, therefore, a system of four to eight processing elements would contain two physical domains. Activities (enabled instructions) are assigned to physical domains based on their context, where each entry into a loop or procedure creates a new context. The test program used here contains one loop;

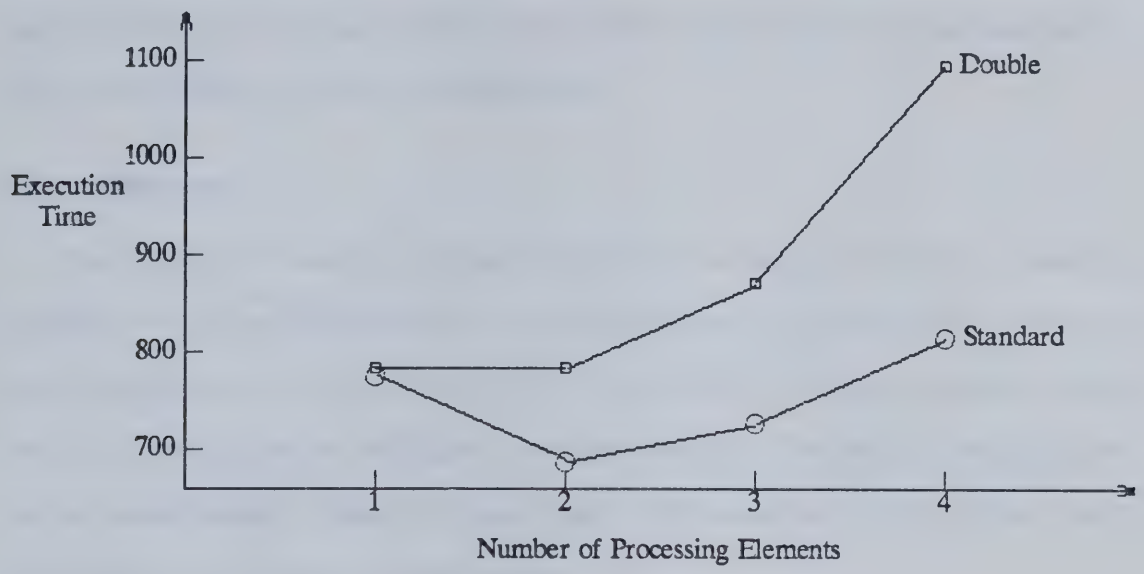



**Figure 4.5 Id Execution Time with Doubled Bus Time**

therefore all statements outside the loop are assigned to context one and all activities within the loop are assigned to context two. Because of this program organization, no activity would ever be assigned to other than the first two physical domains. The addition of a second physical domain would result in increased token travel time with no resultant savings in execution time.

Speed-up, efficiency and average execution time per instruction executed will later be compared to those of the MIT machine.

Simulation results reported by Gostelow and Thomas [GoT80] show execution times of different test programs (their figure 9). In four of the five test programs simulated, the addition of more processing elements produces a reduction in execution time of an asymptotic form. The fifth problem, an iterative fast Fourier transform displays a curve similar to the results shown here in Figure 4.3. An initial reduction of execution time occurs but that is followed by an increase which surpasses the serial execution time. Since these two algorithms (fast Fourier transform and sum-

mation of trapezoids) are similar, the results suggest that this class of algorithms does not map well to the Id machine configuration. These results deserve further investigation to determine what class of algorithms best fits the Id machine configuration, but since automated language compilers are not available this work will not be attempted here.

### 4.2.2. MIT Design

The standard MIT machine is defined to include five functional units. In order to investigate the influence of system configuration on machine performance the memory unit has been partitioned into progressively smaller units, called cell blocks. Each cell block can enable an instruction, thus allowing concurrent enabling of instructions in memory. The number of arbitration units has been correspondingly increased to remain equal to the number of cell blocks, thus allowing each cell block direct access to the operation unit.

Experiments were conducted on five variations of the standard MIT machine with the size of memory cell blocks varying from 32 instructions per cell block to one instruction per cell block. Table 4.5 summarizes the allocation of the test program (nineteen instructions) to the cell blocks within memory. The number of cell blocks increased from one to nineteen.

Resulting execution times are shown in Figure 4.6. Execution time is reduced significantly by the first three memory partitions, but little gain is realized after that. The last memory partition produces the lower limit of one instruction per cell block, where all instructions could theoreti-

**Table 4.5 MIT Allocation of Test Program to Memory**

| MIT Allocation of Test Program to Memory | | |
|---|---|---|
| | Allocation of Instructions to Cell Blocks | |
| Partitioned Cell Block Size | Number of Cell Blocks | Instructions per Block |
| 32 | 1 | 19 |
| 16 | 2 | 16,3 |
| 8 | 3 | 8,8,3 |
| 4 | 5 | 4,4,4,4,3 |
| 2 | 10 | 2,2...2,1 |
| 1 | 19 | 1,1...1,1 |

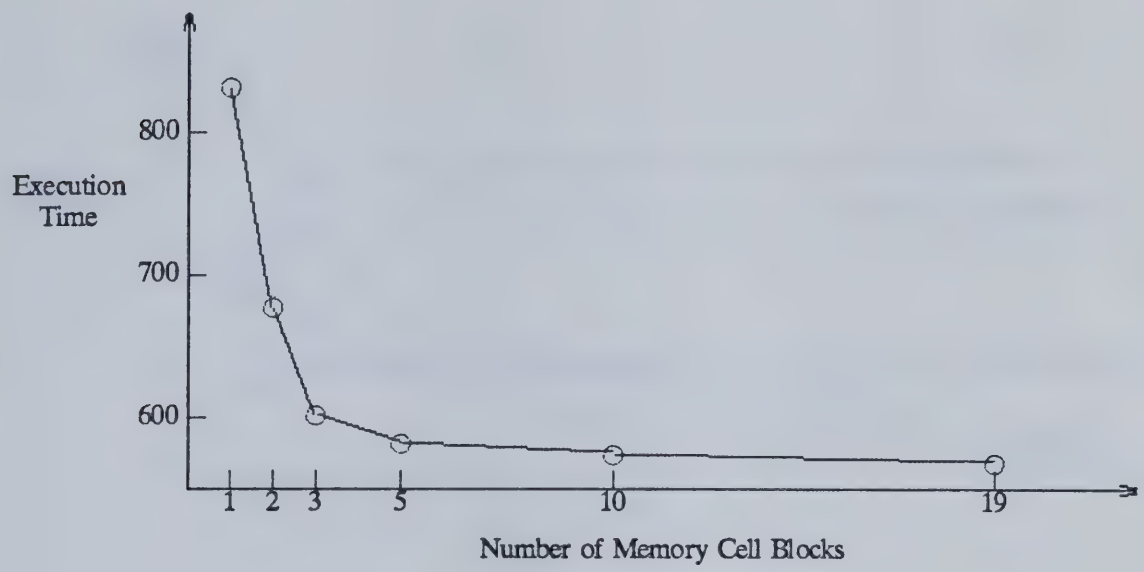



**Figure 4.6 MIT Execution Time**

cally execute simultaneously. Since decreasing the size of memory to this extreme is ineffective, data dependencies and high utilization of other functional units must be preventing further gains, their utilization (see Figure 4.7) and average queue wait time (see Table 4.6) should be examined. The operation unit could be the limiting factor since it exhibits the highest utilization and its average queue wait time has increased as a result of memory partitioning.

The operation unit can be partitioned by function into two units: an arithmetic unit and a boolean/control unit. The execution times resulting from this partition are shown in Figure 4.8. Performance is not improved until the cell block size reaches four instructions per block, and then the improvement is small. The average queue wait times are shown in Table 4.7. Comparison with Table 4.6 shows that the operation unit average queue wait time has not changed significantly but the control network average queue wait time has increased from three to five time units. If analysis where to proceed further, a next step would be to partition the control network by func-

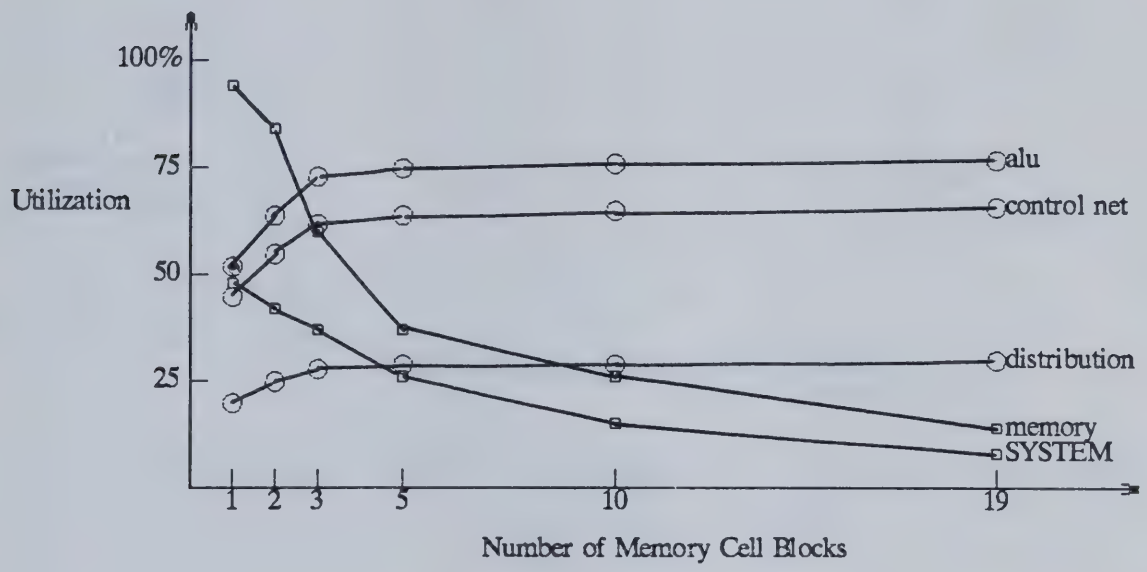



**Figure 4.7 MIT Utilization**

**Table 4.6 MIT Average Queue Wait Time**

| MIT Average Queue Wait Time | | | | | | |
|---|---|---|---|---|---|---|
| | Number of Cell Blocks | | | | | |
| Functional Unit | 1 | 2 | 3 | 5 | 10 | 19 |
| memory | 20 | 8 | 4 | 2 | 2 | 2 |
| operation | 1 | 3 | 2 | 3 | 3 | 3 |
| control network | 3 | 3 | 3 | 3 | 3 | 3 |
| distribution network | 1 | 1 | 1 | 1 | 1 | 1 |
| arbitration network | 0 | 0 | 0 | 0 | 0 | 0 |

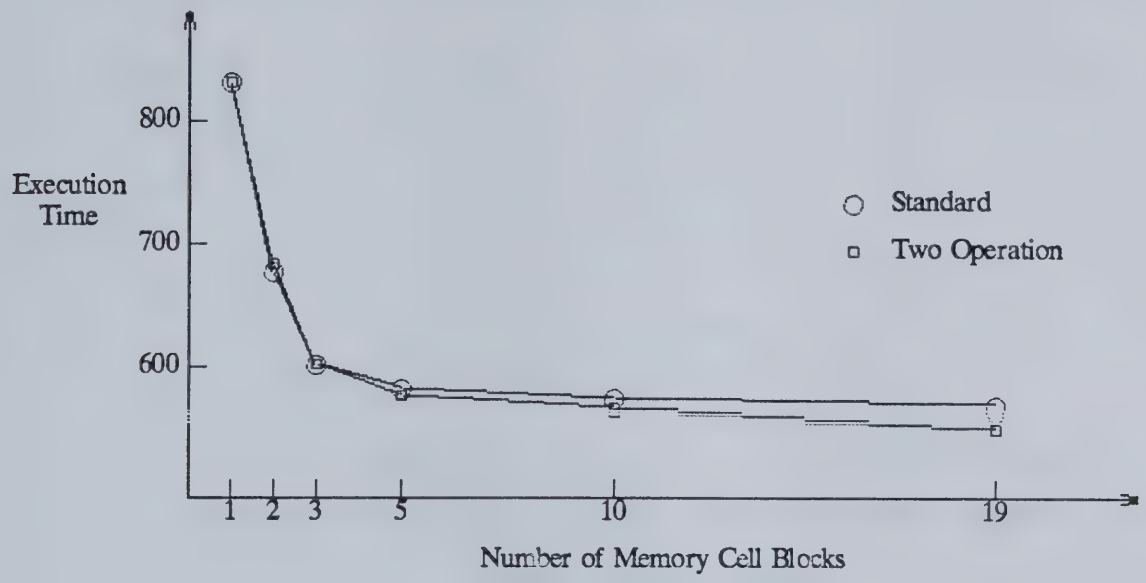



**Figure 4.8 MIT Execution Time with Two Operation Units**

**Table 4.7 MIT Average Queue Wait Time with Two Operation Units**

| MIT Average Queue Wait Time with Two Operation Units | | | | | | |
|---|---|---|---|---|---|---|
| | Number of Cell Blocks | | | | | |
| Functional Unit | 1 | 2 | 3 | 5 | 10 | 19 |
| memory | 20 | 9 | 5 | 2 | 2 | 2 |
| operation unit 1 | 1 | 1 | 1 | 3 | 2 | 3 |
| operation unit 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| control network | 4 | 5 | 5 | 4 | 5 | 5 |
| distribution network | 1 | 1 | 1 | 1 | 1 | 1 |
| arbitration network | 0 | 0 | 0 | 0 | 0 | 0 |

tion or replicate it in order that each operation unit has a dedicated control network.

Memory cell block utilization is shown in Figure 4.9. There is a wide range between highest and lowest cell block utilization. One of the reasons for this range is that instructions within the program iteration loop are executed repeatedly, thus producing a high utilization in that cell block compared to other cell blocks. Perhaps more consistent utilization could be attained by a compiler allocation of instructions to cells with an even distribution of loop and non-loop instructions to cell

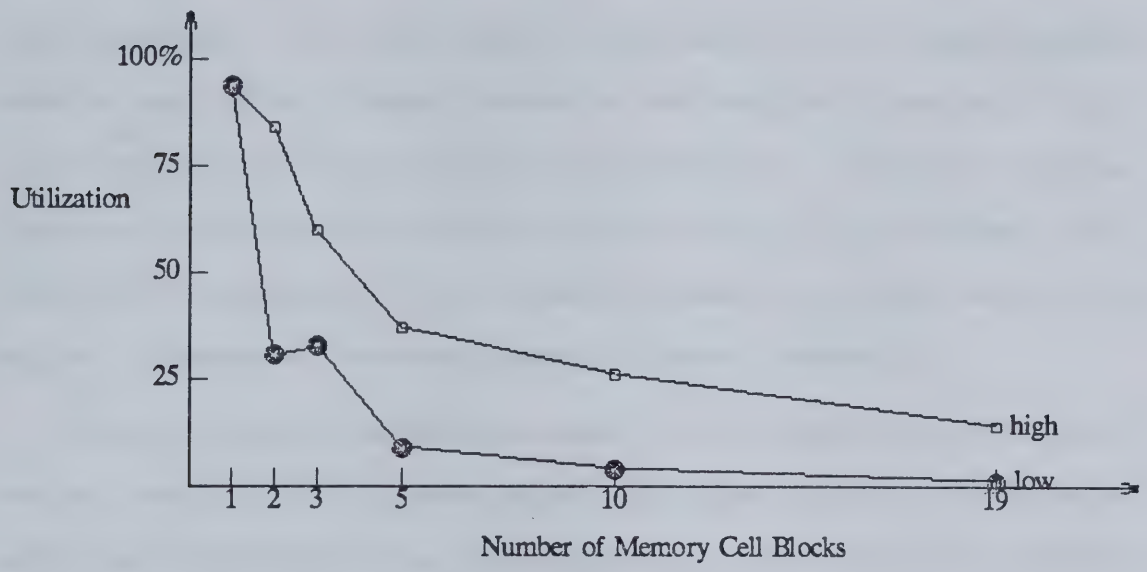



**Figure 4.9 MIT Memory Utilization**

blocks. The Dennis suggestion [DeM75] of a cache memory where instructions are fetched into a cell block upon the arrival of a token would also be a solution but this would involve more time and resources to implement. Cell block utilization is also influenced by the number of instructions allocated to a cell block. As was shown in Table 4.5, the first memory partition resulted in two cell blocks containing sixteen and three instructions respectively. That particular allocation produced a low utilization which becomes an anomaly in Figure 4.9.

No data driven simulation of the MIT machine has been reported in the literature. The Dennis/MIT group have studied a prototype machine which would emulate various dataflow architecture proposals, but results have not been published [DBL80].

### 4.2.3. A Comparison

Insights can be gained by comparing the results of the two machines. The Id machine language program contains thirty-one instructions and the simulation performed sixty-seven

instruction executions. The MIT machine language program is nineteen instructions and the simulation performed forty-four instruction executions. The Id program is much longer because of the unfolding interpreter. The program requires six loop operations and five iteration operations (within the loop), and also all results of decisions are sent through switch operations. The MIT program requires none of the above operations since it employs gates on instructions rather than separate switch operators and has no loop or iteration operators. On the other hand an MIT instruction can specify only one or two destinations; therefore extra control instructions are required. By contrast, the Id instruction can specify any number of destinations.

Because of the use of acknowledgement signals in the MIT machine the instruction enable function is heavily used (131 activations to execute forty-four instructions). The Id machine employs unique activity names rather than acknowledgement signals and the sorter (instruction enable function) unit is less busy (100 activations to execute sixty-seven instructions). The Id machine has more instructions to execute but as can be seen from Figure 4.10, it requires less time to process each.

Speed-up of the two machines is shown in Figure 4.11. Speed-up is the ratio of execution time of the standard system to execution time of the parallel system. Without accurate functional unit execution times and complete machine descriptions, it cannot be concluded that one machine is better than the other. It can be seen that the approaches of the two machines produce radically different results and require further investigation to determine the causes.

An overall measure of system performance is efficiency (ratio of execution time of the standard system to total resource-time usage of the system). To further explain, a program can be executed by the standard system (composed of $N$ functional units) in time $T$. Another system with parallel functional units ($n$ functional units) can execute the same problem in time $t$. A comparison of $N*T$ to $n*t$ will indicate how well resources *and* time are used. Efficiency is an attempt to compare what has been gained to the increased cost of resources. Efficiency of the two machines is shown in Figure 4.12. The results are similar and perhaps indicate a general characteristic of the

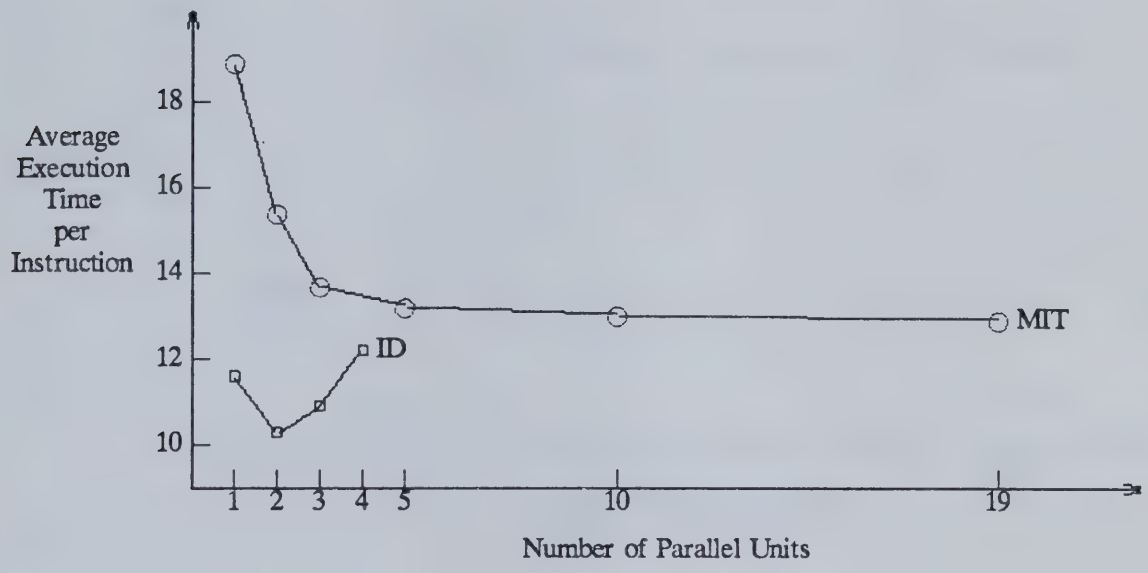



**Figure 4.10 Average Execution Time per Instruction Executed**

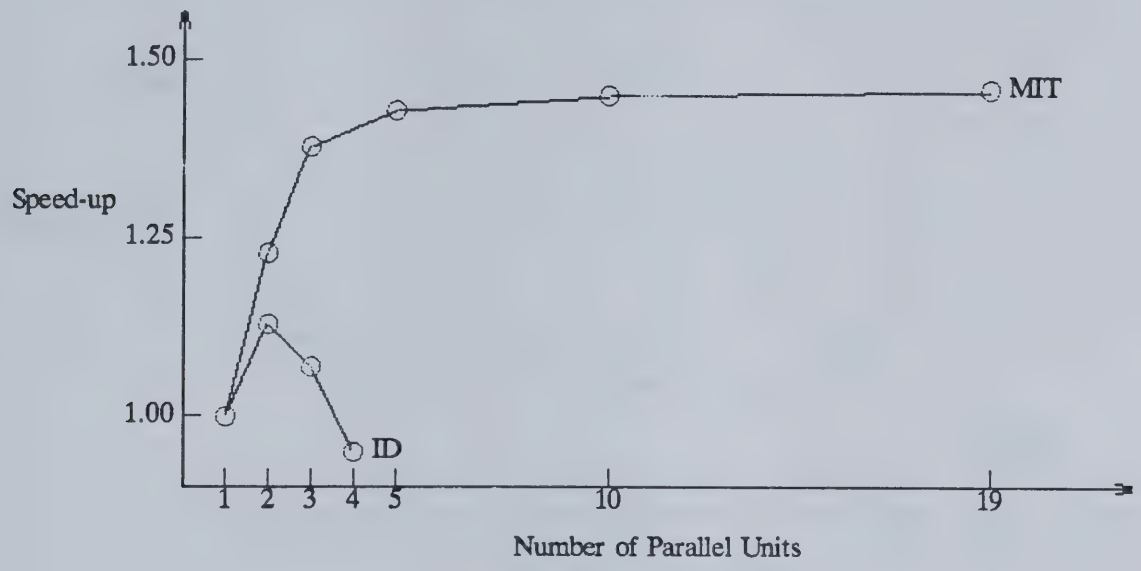



**Figure 4.11 Speed-Up in Execution Time**

performance of dataflow computer architectures. Functional units in a dataflow architecture are not always identical as is implied in the definition of efficiency above, thus efficiency is a gross measure when applied to the evaluation of dataflow architectures. Efficiency is a more appropriate measure in a multiprocessor system where all processing elements are identical. For this reason, efficiency is not recommended for use in further dataflow architecture evaluations.

A major limitation of these results is that the investigation was not carried out with a wide variety of test programs. This is reasonable only with automated program development tools including high level language compilers. Those resources are not currently available.

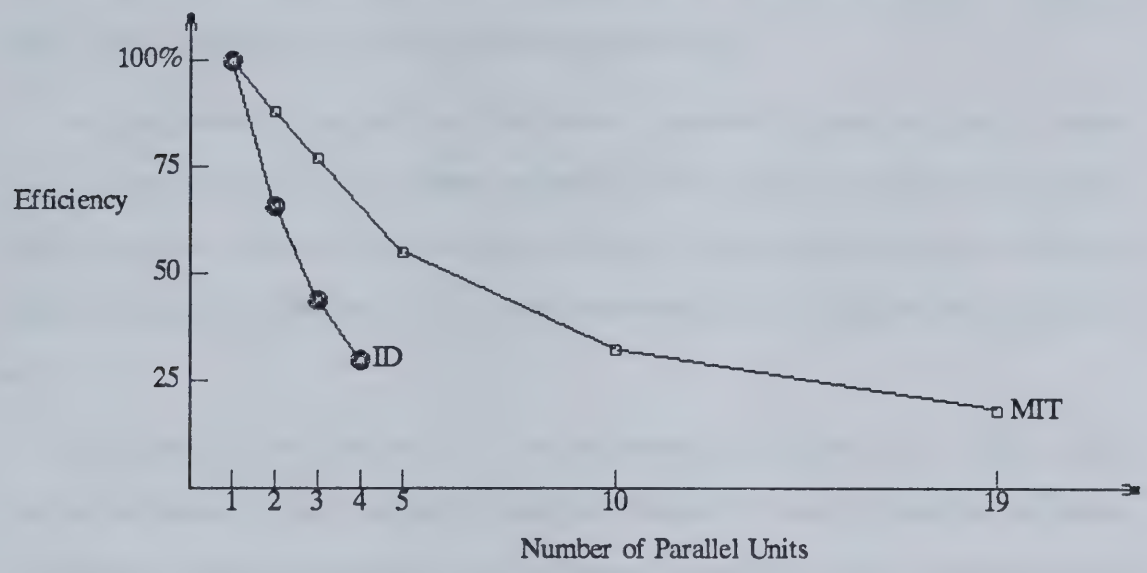



**Figure 4.12 Efficiency of MIT and Id Designs**

## 4.3. Comments on $S_A^*$ and the Simulation Facility

Although $S_A^*$ does not directly map to dataflow architectures as well as the architecture description language ADL [Leu79], it does provide the constructs to accomplish a valid description. For example, an ADL module with specified input and output ports, composed of submodules whose behavior is described using expressions can directly represent a dataflow functional unit. An equivalent description in $S_A^*$ uses a mechanism composed of procedures which accept only input parameters and terminate after activating and sending parameters to another mechanism. Other constructs in $S_A^*$, *call, signal and await,* allow synchronization between processes which is not required in a strict dataflow architecture. But obviously dataflow architectures will evolve such that control flow and dataflow characteristics are integrated to make the best use of both models; therefore these synchronizing constructs are necessary. For example, ADL, designed specifically for packet communication systems, includes capabilities to define functions on data streams and to

protect shared data using monitors.

The queue is used often as a data structure in dataflow architectures therefore it would prove useful to add a queue data type and appropriate operations to $S^*_A$.

As was claimed by the developer, the simulator command language was easy to expand. Also, changes made to add data structures and modify internal operation of the simulator were not difficult to achieve. Almost all changes were confined to the simulator with only minor definition changes required in the compiler. The modularization of the simulator program proved most helpful.

A deficiency of the simulator is the lack of flexibility of the preprocessor which expands identical mechanisms. These mechanisms cannot be referenced by subscript, therefore one particular mechanism cannot be selected from a set of identical mechanisms, instead references are made to the generic name and the first free mechanism is referenced. The ability to reference a particular mechanism should be present and should be similar to an array element reference where the index can be a variable. This facility was required in the descriptions used in this study and was accomplished using a text editor to replicate mechanisms and change names and references to these names. Also, the ability to expand identical systems would be useful.

Some other minor deficiencies of the simulator are lack of real arithmetic, synonyms are limited to a maximum of thirty-two bits and private variables cannot be uniquely identified from outside their scope, therefore initial loading of data before simulation is tricky. Because many $S^*_A$ constructs are not implemented and some construct implementations do not work properly (type, constant, parallel statements), $S^*_A$ descriptions are not as easy to construct as they might be. Also, the time modification presented in this study does not allow a *call* to a public procedure or the use of *await*.

Although no error messages are produced, compiling an $S^*_A$ description is relatively easy, since a file is produced providing enough data to locate the error. Although the simulator itself is not of production quality, it attempts to provide different levels of debugging information. Since

many simulator operations are required to execute one $S_A^*$ statement, the amount of debugging information produced is often overwhelming. On the other hand, a higher debugging level fails to identify the executing $S_A^*$ statement. Debugging the test program itself is often the biggest problem since no automated tools are available. Errors in machine language programs must be found by tracing data through the simulation and examining computation results.

# CHAPTER 5

## Conclusions

The major contribution of this study is the development of a methodology for the evaluation of dataflow computer architecture designs. Because dataflow designs are non-traditional and usually involve many processing units, design and evaluation is a complex task. This complexity demands a structured and systematic design process which utilizes automated tools. An automated design facility at Carnegie-Mellon University, ISPS, has been used for real evaluation and design problems. The backbone of that facility is the use of a formal language to describe the architecture designs and a data-driven simulator. Although those two tools are fundamental to the evaluation process presented here, parameters and performance measures appropriate to the class of dataflow computer architectures are chosen.

Within the dataflow model of computation, operations can be performed concurrently and are sequenced only by the availability of data operands. Because of this possible concurrency, one of the most important measures for evaluation is execution time. Execution time should be investigated under the influence of architecture configuration (number and function of processing elements), component execution times and types of problem. Besides execution time, utilization and average wait time can provide insight into workload balance. Speed-up provides a comprehensive measure of relative performance, but efficiency seems to be too gross a measure to be used for dataflow architecture evaluation.

This study contributes to past work on the $S_A^*$ design environment by extending its previous use to the class of dataflow architectures. The formal language used in this study, $S_A^*$ [Das81], is a general purpose architecture description language whose facilities for concurrent execution of processes make it amenable to dataflow architectures.

The $S_A^*$ simulator [Mak82] has been modified and extended to include capabilities essential to dataflow architecture evaluation. While the simulator provided rudimentary statistics collection facilities, extensions were required to compute execution time of test programs, busy time of processing units and wait time. Because the simulator did not incorporate any concept of time, significant modifications were required. By associating execution times with procedures within processes, the execution time of a test program can be determined. Previously the simulator imitated concurrency by executing all active processes in a round robin order of equal priorities. Now the order of process execution is based on earliest time called. In the dataflow model, processes are initiated asynchronously by the arrival of tokens, therefore an appropriate call mechanism is needed in the simulator. Modifications were made to achieve parameter binding at the time of the call, in order that the calling process may continue execution without waiting for allocation or completion of the called process. Further suggestions for improvements to $S_A^*$ and the simulator were detailed in Chapter 4.

The practicality of the proposed evaluation methodology has been tested on two dataflow architecture designs. Sufficient experimental results have been presented to demonstrate the applicability of the method to divergent designs within this class. Although data-driven simulation can provide many valuable insights, a major difficulty is the choice of test programs used as input to the simulator. The experiments presented here have been limited by the necessity to manually code in machine language each program for each machine. Besides the amount of time required to produce machine language programs, other difficulties are created. Humans tend to make mistakes, increasing the amount of debugging time and introducing a reluctance to attempt complex problems. A real evaluation facility must provide high level language compilers so that a variety, in application, algorithm, and size of test programs can be produced quickly and correctly.

Further research to determine whether different algorithms optimally fit different architectures is important. Positive results would call for an investigation on a more abstract level to determine if particular architecture behaviors (e.g. dynamic node replication versus

acknowledgement signals) produce consistently better performance on particular algorithms.

The example tests presented in Chapter 4 investigate only a few architecture characteristics. The evaluation methodology could be used to investigate many other attributes such as machine language representation and primitive data structures.

## References


[AGP78] Arvind, K. P. Gostelow and W. Plouffe, The (Preliminary) Id Report: An Asynchronous Programming Language and Computing Machine, Technical Report #114, Department of Information and Computer Science, University of California, Irvine, 1978.

[AKP80] Arvind, V. Kathail and K. Pingali, A Dataflow Architecture With Tagged Tokens, MIT/LCS/TM-174, Massachusetts Institute of Technology, 1980.

[Bac78] J. Backus, Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, *Comm. ACM 21*, 8 (1978), 613-641.

[Bae80] J. Baer, *Computer Systems Architecture*, Computer Science Press, 1980.

[BaS77] M. R. Barbacci and D. P. Siewiorek, Evaluation of the CFA Test Programs via Formal Computer Descriptions, *IEEE Computer 10*, 10 (1977), 36-43.

[Bar79] M. R. Barbacci, Instruction Set Processor Specifications (ISPS): The Notation and its Applications, CMU-CS-79-123, Department of Computer Science, Carnegie-Mellon University, 1979.

[BeN71] C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, 1971.

[Das81] S. Dasgupta, $S_A^*$: A Language For Describing Computer Architectures, in *Computer Hardware Description Languages and Their Applications*, M. Bruer and R. Hartenstein (ed.), North-Holland Publishing Company, 1981, 65-78.

[Das82] S. Dasgupta, Computer Design and Description Languages, in *Advances in Computers Volume 21*, Marshall C. Yovits (ed.), Academic Press, 1982, 91-154.

[Das83] S. Dasgupta, *A Definition of the Architecture Description Language* $S_A^*$, Department of Computer Science, University of Southwestern Louisiana, Lafayette, Louisiana, 1983.

[DaD80] A. L. Davis and P. J. Drongowski, Dataflow Computers: A Tutorial and Survey, UUCS-80-109, Department of Computer Science, University of Utah, Salt Lake City, Utah, 1980.

[DeM75] J. B. Dennis and D. P. Misunas, A Preliminary Architecture for a Basic Data-Flow Processor, *Procedings of the 2nd International Symposium on Computer Architecture*, 1975, 126-132.

[DBL80] J. B. Dennis, G. A. Boughton and C. K. C. Leung, Building Blocks for Data Flow Prototypes, *The Seventh Annual Symposium on Computer Architecture*, 1980, 1-8.

[Den80] J. B. Dennis, Data Flow Supercomputers, *IEEE Computer 13*, 11 (1980), 48-56.

[DLM80] J. B. Dennis, C. K. Leung and D. P. Misunas, A Highly Parallel Processor Using a Data Flow Machine Language, Computation Structures Group Memo 134-2, Massachusetts Institute of Technology, 1977, revised 1980.

[FuB77] S. H. Fuller and W. E. Burr, Measurement and Evaluation of Alternative Computer Architectures, *IEEE Computer 10*, 10 (1977), 24-35.

[GoT80] K. P. Gostelow and R. E. Thomas, Performance of a Simulated Dataflow Computer, *IEEE Transactions on Computers c-29*, 10 (1980), 905-919.

[Hay78] J. P. Hayes, *Computer Architecture and Organization*, McGraw-Hill, 1978.

[Jen81] S. Jennings, *Petri Net Models of Program Execution in Data Flow Environments*, PhD Diss., Iowa State University, Ames, Iowa, 1981.

[Kel74] R. M. Keller, Towards a Theory of Universal Speed-Independent Modules, *IEEE Transactions on Computers C-23*, 1 (1974), 21-33.

[Lee80] R. B. Lee, Empirical Results on the Speed, Efficiency, Redundancy and Quality of Parallel Computations, *Proceedings of the 1980 International Conference on Parallel Processing*, 1980, 91-100.

[Leu79] C. K. C. Leung, ADL: An Architecture Description Language for Packet Communication Systems, *Proceedings of the 1979 International Symposium on Hardware Description Languages and their Applications*, Palo Alto, California, 1979, 6-13.

[Mak82] D. D. Makarenko, Simulating Computer Architectures Using an Architecture Description Language, TR82-5, Department of Computing Science, The University of Alberta, Edmonton, Alberta, 1982.

[Mas80] Massachusetts Institute of Technology, *Laboratory for Computer Science Progress Report 17*, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1979-June 1980.

[Mey76] S. C. Meyer, An Analytic Approach to Performance Analysis for a Class of Data Flow Processors, *Proceedings of the 1976 International Conference on Parallel Processing*, 1976, 106-115.

[Mis76] D. P. Misunas, Performance Analysis of a Data-Flow Processor, *Proceedings of the 1976 International Conference on Parallel Processing*, 1976, 100-105.

[Pla76] A. Plas et al., LAU System Architecture: A Parallel Data-Driven Processor Based on Single Assignment, *Proceedings of the 1976 International Conference on Parallel Processing*, 1976, 293-302.

[SCH77] J. C. Syre, D. Comte and N. Hifdi, Pipelining, Parallelism and Asynchronism in the LAU System, *Proceedings of the 1977 International Conference on Parallel Processing*, 1977, 87-92.

[TrL82] P. C. Treleaven and I. G. Lima, Japan's Fifth-Generation Computer Systems, *IEEE Computer 15*, 8 (1982), 79-88.

[TBH82] P. C. Treleaven, D. R. Brownbridge and R. P. Hopkin, Data-Driven and Demand-Driven Computer Architecture, *ACM Computing Surveys 1*, 3 (1982), 93-143.

[VBH81] A. Van Dam, M. Barbacci, C. Halatsis, J. Joosten and M. Letheran, Simulation of a Horizontal Bit-Sliced Processor Using the ISPS Architecture Simulation Facility, *IEEE Transactions on Computers c-30*, 7 (1981), 513-519.

[WaG79] I. Watson and J. Gurd, A Prototype Data Flow Computer with Token Labeling, *AFIPS Procedings of the National Conference*, 1979, 623-628.

# APPENDIX A

## S*A Description of Id Machine

```
/* Id (Irvine Dataflow) Architecture Description
   March 1983
*/

sys id;


        glovar  no_pe, no_pd, pe_per_pd : seq[31 .. 0] bit;                    /* parameter */
        glovar  q_size : seq[31 .. 0] bit;                              /* parameter */
        glovar  lring : array[8 .. 15] of seq[31 .. 0] bit;             /* parameter 8 to 8*(no_pe+1)-1 */
        glovar  rring : array[8 .. 15] of seq[31 .. 0] bit;             /* parameter as above */
        glovar  q_mem_response  : array[0 .. 99] of seq[31 .. 0] bit;
        glovar  q_alu   : array[0 .. 99] of seq[31 .. 0] bit;
        glovar  q_out   : array[0 .. 99] of seq[31 .. 0] bit;
        glovar  q_left  : array[0 .. 99] of seq[31 .. 0] bit;
        glovar  q_right : array[0 .. 99] of seq[31 .. 0] bit;
        glovar  shared_q_left_rear      : seq[31 .. 0] bit;
        glovar  shared_q_right_rear     : seq[31 .. 0] bit;

mech ring;

        privar count : seq[31 .. 0] bit;

        priv proc all_shift()
                count := 0;
                act ring_interfacex001.read();
                return
        endproc

        proc parameters();      /*set parameters and activate simulation */
                no_pe := 1;
                no_pd := 1;
                pe_per_pd := 1;
                q_size := 99;
                act ring_interfacex001.start_up();      /* activate each pe */
                exit
        endproc         /* parameters */


        proc shift();           /* synchronizes access to the two token busses */

                count := count + 1;
                if (count == no_pe) => call all_shift()
                fi;
                exit
        endproc         /* shift */
endmech                 /* ring */
; init ring.parameters()
```

```
sys physical_domain1;
        sys memory1;

                mech mem_read1;

                        privar mem : array[0 .. 310] of seq[31 .. 0] bit;
                        privar table : array[0 .. 40] of seq[31 .. 0] bit;
                        privar u : seq[31 .. 0] bit;
                        privar c : seq[31 .. 0] bit;
                        privar s : seq[31 .. 0] bit;
                        privar iter : seq[31 .. 0] bit;
                        privar arg_address : seq[31 .. 0] bit;
                        privar ptr : seq[31 .. 0] bit;
                        privar rear : seq[31 .. 0] bit;
                        privar no_dests : seq[31 .. 0] bit;
                        privar caller : seq[31 .. 0] bit;
                        privar k : seq[31 .. 0] bit;

                        proc get_code(<u,<c,<s,<iter,<arg_address,<caller);

                                ptr := table[s];
                                no_dests := mem[ptr+5];
                                k := 0;
                                while (k <= (5+(3 * no_dests))) do
                                        if (rear == q_size) => rear := 0
                                                else => rear := rear + 1
                                        fi;
                                        q_mem_response[rear] := mem[ptr + k];
                                        k := k + 1
                                od;

                                if (caller == 1) => act receivex001.mem_response(<no_dests,<u,<c,<s,<iter,<arg_address)
                                fi;
                                exit
                        endproc         /* get_code */
                endmech         /* mem_read1 */
        endsys;         /* memory1 */
```

```
sys pex001;
        glovar pe_id : seq[31 .. 0] bit;
        glovar lring_ptr,rring_ptr : seq[31 .. 0] bit;
        glovar fast_mem : array[0 .. 400] of tuple      u : seq[31 .. 0] bit;
                                                        c : seq[31 .. 0] bit;
                                                        s : seq[31 .. 0] bit;
                                                        iter : seq[31 .. 0] bit;
                                                        port : seq[31 .. 0] bit;
                                                        tvalue : seq[31 .. 0] bit;
                                                        ptr : seq[31 .. 0] bit
                                                endtup;
```

```
mech sortx001;
        privar count, u,c,s,iter,port,tvalue,no_inputs,i,k,
                pointer, oldpointer,first_ptr : seq[31 .. 0] bit;

        proc sorter(<u,<c,<s,<iter,<tvalue,<port,<no_inputs);

        /*insert token in fast_mem */
                i := i + 1;
                first_ptr := i;

                fast_mem.u[i] := u;
                fast_mem.c[i] := c;
                fast_mem.s[i] := s;
                fast_mem.iter[i] := iter;
                fast_mem.port[i] := port;
                fast_mem.tvalue[i] := tvalue;
                fast_mem.ptr[i] := -1;
                count := 1;
        /* find matching activities in fast_mem */
        if (no_inputs ¬= count) => do
                k := 1;
                while (k < i and (¬ (fast_mem.u[k] == u
                                and fast_mem.c[k] == c
                                and fast_mem.s[k] == s
                                and fast_mem.iter[k] == iter))) do
                        k := k + 1
                od;
                if (k == i) => pointer := -1
                        else => do
                                pointer := k;
                                first_ptr := k;
                /* count the number of matching activites */
                                while (pointer ¬= -1) do
                                        count := count + 1;
                                        oldpointer := pointer;
                                        pointer := fast_mem.ptr[pointer]
                                od;
                                fast_mem.ptr[oldpointer] := i
                        od
                fi
                od
        fi;
        /* if all activities are here,  send to code fetch */
                if (count == no_inputs) =>
                        act fetchx001.code(<u,<c,<s,<iter,<first_ptr)
                fi;
                exit
        endproc         /* sorter */
endmech         /* sortx001 */
```

```
mech fetchx001;

        privar u : seq[31 .. 0] bit;
        privar c : seq[31 .. 0] bit;
        privar s : seq[31 .. 0] bit;
        privar iter : seq[31 .. 0] bit;
        privar arg_address : seq[31 .. 0] bit;

        proc code(<u,<c,<s,<iter,<arg_address);

                act mem_read1.get_code(<u,<c,<s,<iter,<arg_address,<pe_id);
                exit
        endproc /*code */
endmech         /* fetchx001 */

mech receivex001;

        privar u : seq[31 .. 0] bit;
        privar c : seq[31 .. 0] bit;
        privar s : seq[31 .. 0] bit;
        privar iter : seq[31 .. 0] bit;
        privar no_dests : seq[31 .. 0] bit;
        privar arg_address : seq[31 .. 0] bit;
        privar k, rear, front : seq[31 ..0] bit;

        proc mem_response (<no_dests,<u,<c,<s,<iter,<arg_address);

                k := 0;
                while (k <= (5 + 3 * no_dests)) do
                        if (rear == q_size) => rear := 0
                                else => rear := rear + 1
                        fi;
                        if (front == q_size) => front := 0
                                else => front := front + 1
                        fi;
                        q_alu[rear] := q_mem_response[front];
                        k := k + 1
                od;
                act alux001.exec(<no_dests,<u,<c,<s,<iter,<arg_address);
                exit
        endproc /* mem_response */
endmech         /* receivex001 */
```

```
mech alux001;

        privar bottom, top, rear, front, result, temp, opcode, is_constant,
                const_port, constant,flush      : seq[31 .. 0] bit;
        privar no_dests, u, c, s, iter, arg_address, k  : seq[31 .. 0] bit;
        privar arg      : array [0 ..1] of seq[31 .. 0] bit;

        priv proc addq_out(<bottom)
                if (rear == q_size) => rear := 0
                        else => rear := rear + 1
                fi;
                q_out[rear] := bottom;
                return
        endproc

        priv proc deleteq_alu(>top)
                if (front == q_size) => front := 0
                        else => front := front + 1
                fi;
                top := q_alu[front];
                return
        endproc

        proc exec(<no_dests,<u,<c,<s,<iter,<arg_address);

                flush := 0;
                result := 0;                    /* initialize */
                arg[0] := 0;
                arg[1] := 0;

                call deleteq_alu(>temp);
                call deleteq_alu(>opcode);
                call deleteq_alu(>is_constant);
                call deleteq_alu(>const_port);
                call deleteq_alu(>constant);
                call deleteq_alu(>temp);

/* get arguments */
                while (arg_address ¬= -1) do
                        arg[fast_mem.port[arg_address]] := fast_mem.tvalue[arg_address];
                        arg_address := fast_mem.ptr[arg_address]
                od;

                if (is_constant == 1) => arg[const_port] := constant
                fi;
```

```
/* execute opcodes */
                if (opcode == 1) => result := arg[0] + arg[1]          /* + */
                || (opcode == 2) => result := arg[0] - arg[1]          /* - */
                || (opcode == 3) => result := arg[0] * arg[1]          /* * */
                || (opcode == 4) => result := arg[0] / arg[1]          /* / */
                || (opcode == 5) => result := arg[0] * arg[0]          /* SQUARED */
                || (opcode == 17) => if (arg[0] <= arg[1]) => result := 1
                                     else => result :=0                /* < */
                                     fi
                || (opcode == 19) => if (arg[0] > arg[1]) => result := 1
                                     else => result :=0                /* > */
                                     fi
                || (opcode == 20) => if (arg[0] == arg[1]) => result := 1
                                     else => result :=0                /* EQ */
                                     fi
                || (opcode == 40) => result := constant                /* pass constant unchanged */

                || (opcode == 50) => do result := arg[0];       /* Switch */
                                     if (arg[1] == 1) => do
                                             flush := no_dests - constant;
                                             no_dests := constant
                                             od
                                     else => do
                                             no_dests := no_dests - constant;
                                             k := 1;
                                             while (k <= 3*constant) do
                                                     call deleteq_alu(>temp);
                                                     k := k + 1
                                             od
                                     od
                                     fi
                                    od
                || (opcode == 100) => do u := u + 1;           /* L loop operator */
                                      c := constant;
                                      iter := 1;
                                      result := arg[0]
                                      od
                || (opcode == 101) => do u := u - 1;           /* L -1 */
                                      c := constant;
                                      result := arg[0]
                                      od
                || (opcode == 200) => do iter := iter + 1;     /* 'D' Iteration operator */
                                      result := arg[0]
                                      od
                || (opcode == 201) => do iter := 1;            /* 'D' reset iteration */
                                      result := arg[0]
                                      od
```

```
|| (opcode == 300) =>  do result := arg[0];      /* output */
                        print(result);
                        break(9999)
                       od

fi;
```

```
        /* send tokens to output */
                if (no_dests > 0) => do
                        call addq_out(<no_dests);
                        call addq_out(<u);
                        call addq_out(<c);
                        call addq_out(<s);
                        call addq_out(<iter);
                        call addq_out(<result);
                        k := 1;
                        while (k <= 3 * no_dests) do
                                call deleteq_alu(>temp);
                                call addq_out(<temp);
                                k := k + 1
                        od
                        od
                fi;

                if (flush > 0) => do
                        k := 1;
                        while (k <= 3 * flush) do
                                call deleteq_alu(>temp);
                                k := k + 1
                        od
                        od
                fi;

                if (no_dests > 0) =>
                        act outputx001.make_tokens()
                fi;
                exit
        endproc         /* exec */
endmech         /* alux001 */
```

```
mech outputx001;

        privar top, front, bottom, rear_left, rear_right : seq[31 .. 0] bit;
        privar no_dests, u, c, s, iter, result, port     : seq[31 .. 0] bit;
        privar no_inputs, address, x,y, direction, k,temp  : seq[31 .. 0] bit;

        priv proc deleteq_out(>top)
                if (front == q_size) => front := 0
                        else => front := front + 1
                fi;
                top := q_out[front];
                return
        endproc

        priv proc addq_left(<bottom)
                if (rear_left == q_size) => rear_left := 0
                        else => rear_left := rear_left + 1
                fi;
                q_left[rear_left] := bottom;
                return
        endproc
        priv proc addq_right(<bottom)
                if (rear_right == q_size) => rear_right := 0
                        else => rear_right := rear_right + 1
                fi;
                q_right[rear_right] := bottom;
                return
        endproc

        proc make_tokens();

                call deleteq_out(>no_dests);
                call deleteq_out(>u);
                call deleteq_out(>c);
                call deleteq_out(>temp);
                call deleteq_out(>iter);
                call deleteq_out(>result);
                k := 1;
                while (k <= no_dests) do
                        call deleteq_out(>s);
                        call deleteq_out(>port);
                        call deleteq_out(>no_inputs);
```

```
/* calculate physical address */
        x := s;
        while (x > pe_per_pd) do
                x := x - pe_per_pd
        od;
        y := u;
        while (y > no_pd) do
                y := y - no_pd
        od;

        address := (pe_per_pd * (y - 1)) + x;
        if (address == pe_id) =>
                act sortx001.sorter(<u,<c,<s,<iter,<result,<port,<no_inputs)
        else => do
                if (address > pe_id) =>
                        if ((address - pe_id) < (no_pe / 2)) =>
                                direction := 1
                        else => direction := 0
                        fi
                else =>
                        if ((pe_id - address) < (no_pe / 2)) =>
                                direction := 0
                        else => direction := 1
                        fi
                fi;
                if (direction == 0) => do
                        shared_q_left_rear := rear_left;
                        call addq_left(<address);
                        call addq_left(<u);
                        call addq_left(<c);
                        call addq_left(<s);
                        call addq_left(<iter);
                        call addq_left(<result);
                        call addq_left(<port);
                        call addq_left(<no_inputs);
                        shared_q_left_rear := rear_left
                        od
```

```
                                else => do
                                        shared_q_right_rear := rear_right;
                                        call addq_right(<address);
                                        call addq_right(<u);
                                        call addq_right(<c);
                                        call addq_right(<s);
                                        call addq_right(<iter);
                                        call addq_right(<result);
                                        call addq_right(<port);
                                        call addq_right(<no_inputs);
                                        shared_q_right_rear := rear_right
                                        od
                                fi
                                od


                        fi;
                        k := k + 1
                od;     /* while k <= no_dests */
                exit
        endproc         /* make_tokens */
endmech         /* outputx001 */
```

```
mech ring_interfacex001;

        privar front_left, front_right : seq[31 .. 0] bit;
        privar k, temp, top      : seq[31 .. 0] bit;

        priv proc deleteq_left(>top)
                if (front_left == q_size) => front_left := 0
                        else => front_left := front_left + 1
                fi;
                top := q_left[front_left];
                return
        endproc

        priv proc deleteq_right(>top)
                if (front_right == q_size) => front_right := 0
                        else => front_right := front_right + 1
                fi;
                top := q_right[front_right];
                return
        endproc


        proc start_up();
                pe_id := 1;                 /* parameter */
                lring_ptr, rring_ptr := 8 * pe_id;
                act ring_interfacex001.read();
                exit
        endproc         /* start_up */
```

```
proc read();

/* read left token bus */
        /* shift pointer left one slot */
        if (lring_ptr == 8) => lring_ptr := 8 * no_pe
                else => lring_ptr := lring_ptr - 8
        fi;

        /* if matching address, send token to sorter and update slot */
        if (lring[lring_ptr] == pe_id) => do
                act sortx001.sorter(    <lring[lring_ptr + 1],
                                        <lring[lring_ptr + 2],
                                        <lring[lring_ptr + 3],
                                        <lring[lring_ptr + 4],
                                        <lring[lring_ptr + 5],
                                        <lring[lring_ptr + 6],
                                        <lring[lring_ptr + 7]);
                lring[lring_ptr] := 0
                od
        fi;
        if ((front_left ¬= shared_q_left_rear)
                        and (lring[lring_ptr] == 0)) => do
                k := 0;
                while (k <= 7) do
                        call deleteq_left(>temp);
                        lring[lring_ptr + k] := temp;
                        k := k + 1
                        od
                od
        fi;
```

```
        /* read right token bus */
                /* shift pointer right one slot */
                if (rring_ptr == 8 * no_pe) => rring_ptr := 8
                        else => rring_ptr := rring_ptr + 8
                fi;

                /* if matching address, send token to sorter and update slot */
                if (rring[rring_ptr] == pe_id) => do
                        act sortx001.sorter(    <rring[rring_ptr + 1],
                                                <rring[rring_ptr + 2],
                                                <rring[rring_ptr + 3],
                                                <rring[rring_ptr + 4],
                                                <rring[rring_ptr + 5],
                                                <rring[rring_ptr + 6],
                                                <rring[rring_ptr + 7]);
                        rring[rring_ptr] := 0
                        od
                fi;
                if ((front_right ~= shared_q_right_rear)
                                and (rring[rring_ptr] == 0)) => do
                        k := 0;
                        while (k <= 7) do
                                call deleteq_right(>temp);
                                rring[rring_ptr + k] := temp;
                                k := k + 1
                                od
                        od
                fi;

        /* act ring.shift to signal completion of token bus read */
                act ring.shift();
                exit
        endproc         /* read */
endmech         /* ring_interfacex001 */

endsys;         /* pex001 */
endsys;         /* physical domain */

endsys;         /* id */
```

# APPENDIX B

## S*A Description of MIT Machine

```
/* MIT Dataflow Architecture Description
   March 1983
*/

sys mit;

        glovar f1 :     array[0 .. 75] of seq[31 .. 0] bit;
        glovar f2 :     array[0 .. 75] of seq[31 .. 0] bit;
        glovar f3 :     array[0 .. 75] of seq[31 .. 0] bit;
        glovar f4 :     array[0 .. 75] of seq[31 .. 0] bit;
        glovar f5 :     array[0 .. 75] of seq[31 .. 0] bit;
        glovar f6 :     array[0 .. 75] of seq[31 .. 0] bit;
        glovar f7 :     array[0 .. 75] of seq[31 .. 0] bit;
        glovar f8 :     array[0 .. 75] of seq[31 .. 0] bit;
        glovar f9 :     array[0 .. 75] of seq[31 .. 0] bit;

mech memory;

        privar address, dummy   :       seq[31 .. 0] bit;

proc enable(<address);

        if ((address < 0) or (address >68)) => break(6666)      /*error*/
        fi;

        /* can instruction be enabled? */
        if (f3[address]<f4[address]) => exit    /* awk_received == awk_required? */
        fi;

        if (f2[address] == 01) =>               /* format = 01 */
                if ( ((f2[address+1] == 3) or (f2[address+1] == f3[address+1]))
                                        /* if gate = constant or gate = gate flag */
                        and (f7[address+1] == 1)                        /* operand flag is enabled */
                        and ((f2[address+2] == 3) or (f2[address+2] == f3[address+2]))
                                        /* repeat for second operand */
                        and (f7[address+2] == 1) ) => do

                /* send ( format, opcode, val1, val2, dest1,awk address) */
                        act arbit_net.sort( <f2[address],<f5[address],<f5[address+1],<f5[address+2],
                                <f7[address],<address, <dummy, <dummy, <dummy,<dummy );

                /* disable instruction */
                        f3[address] := 0;                       /* reset awk_rec to 0 */
                        f3[address+1], f3[address+2] := 0;      /* reset gate flags */
                        if (f2[address+1] ¬= 3) => f7[address+1] :=0
                        fi;                                     /* reset value flag if not constant */
                        if (f2[address+2] ¬= 3) => f7[address+2] :=0
                        fi                                      /* reset value flag */
                        od
                fi
```

```
|| (f2[address] == 02) =>                  /* format = 02 */
        if ( ((f2[address+2] == 3) or (f2[address+2] == f3[address+2]))
                                  /* if gate = constant or gate = gate flag */
                and (f7[address+2] == 1) ) => do                          /* operand flag is enabled */

        /* send ( format, opcode, val1, dest1, dest2, awk address) */
                act arbit_net.sort( <f2[address],<f5[address],<f5[address+2],<f7[address],
                        <f7[address+1],<address, <dummy, <dummy, <dummy, <dummy );

        /* disable instruction */
                f3[address] := 0;                 /* reset awk_rec to 0 */
                f3[address+2] := 0;               /* reset gate flags */
                if (f2[address+2] ¬= 3) => f7[address+2] :=0
                fi                                /* reset value flag if not constant */
                od
        fi
|| (f2[address] == 03) =>                  /* format = 03 */
        if ( ((f2[address+1] == 3) or (f2[address+1] == f3[address+1]))
                                  /* if gate = constant or gate = gate flag */
                and (f7[address+1] == 1)                                  /* operand flag is enabled */
                and ((f2[address+2] == 3) or (f2[address+2] == f3[address+2]))
                                  /* repeat for second operand */
                and (f7[address+2] == 1) ) => do

        /* send ( format, opcode, val1, val2, tag1, dest1,awk address) */
                act arbit_net.sort( <f2[address],<f5[address],<f5[address+1],<f5[address+2],
                        <f6[address],<f7[address], <address, <dummy, <dummy, <dummy );

        /* disable instruction */
                f3[address] := 0;                 /* reset awk_rec to 0 */
                f3[address+2] := 0;               /* reset gate flags */
                if (f2[address+1] ¬= 3) => f7[address+1] :=0
                fi;                               /* reset value flag fi not constant */
                if (f2[address+2] ¬= 3) => f7[address+2] :=0
                fi                                /* reset value flag */
                od
        fi
```

```
        || (f2[address] == 06) =>                /* format = 06 */
                if ( ((f2[address+2] == 3) or (f2[address+2] == f3[address+2]))
                                                 /* if gate = constant or gate = gate flag */
                        and (f7[address+2] == 1) ) => do                        /* operand flag is enabled */

                /* send ( format, opcode, c1, tag1, dest1,tag2, dest2, tag3, dest3, awk address) */
                        act arbit_net sort( <f2[address],<f5[address],<f5[address+2],<f6[address],
                                <f7[address],<f6[address+1], <f7[address+1],<f8[address+2],
                                <f9[address+2],<address );
                /* disable instruction */
                        f3[address] := 0;                /* reset awk_rec to 0 */
                        f3[address+2] := 0;              /* reset gate flags */
                        if (f2[address+2] ¬= 3) => f7[address+2] :=0
                        fi                               /* reset value flag if not constant */
                        od
                fi
        fi;
        exit
endproc         /* enable */
endmech         /* memory */
```

```
mech arbit_net;

        privar a1,a2,a3,a4,a5,a6,a7,a8,a9,a10, opcode : seq[31 .. 0] bit;

proc sort (<a1, <opcode, <a3,<a4,<a5,<a6,<a7,<a8,<a9,<a10);

        if (opcode == 1) => act operation.add(<a1, <opcode, <a3,<a4,<a5,<a6,<a7,<a8,<a9,<a10)
        || (opcode == 2) => act operation.sub(<a1, <opcode, <a3,<a4,<a5,<a6,<a7,<a8,<a9,<a10)
        || (opcode == 3) => act operation.mult(<a1, <opcode, <a3,<a4,<a5,<a6,<a7,<a8,<a9,<a10)
        || (opcode == 4) => act operation.div(<a1, <opcode, <a3,<a4,<a5,<a6,<a7,<a8,<a9,<a10)
        || (opcode == 5) => act operation.exp(<a1, <opcode, <a3,<a4,<a5,<a6,<a7,<a8,<a9,<a10)
        || (opcode == 6) => act operation.ident(<a1, <opcode, <a3,<a4,<a5,<a6,<a7,<a8,<a9,<a10)
        || (opcode == 17) => act operation.less(<a1, <opcode, <a3,<a4,<a5,<a6,<a7,<a8,<a9,<a10)
        || (opcode == 16) => act operation.pass(<a1, <opcode, <a3,<a4,<a5,<a6,<a7,<a8,<a9,<a10)
        || (opcode == 300) => act operation.output(<a1, <opcode, <a3,<a4,<a5,<a6,<a7,<a8,<a9,<a10)
        fi;
        exit
endproc         /* sort */
endmech         /* arbit_net */
```

```
mech operation;

        privar format, opcode, val1,val2,dest1,awk,a7,a8,a9,a10 :seq[31 .. 0] bit;
        privar result, i, c1, tag1,tag2,tag3,dest2,dest3 :seq[31 .. 0] bit;


/* operators: add, sub, div, exp, ident, output */

proc add(<format, <opcode, <val1,<val2,<dest1,<awk,<a7,<a8,<a9,<a10);

        if ((format¬= 01) or (opcode ¬= 01)) => do
                print(format);
                print(opcode);
                break(900)
                od
        fi;

        result := val1 + val2;
        act dist_net.data (<dest1,< result,< awk);
        awk := awk +1;
        act control_net.awk (<awk);
        awk := awk +1;
        act control_net.awk (<awk);
        exit
endproc         /* add */

proc sub(<format, <opcode, <val1,<val2,<dest1,<awk,<a7,<a8,<a9,<a10);

        if ((format¬= 01) or (opcode ¬= 02)) => do
                print(format);
                print(opcode);
                break(902)
                od
        fi;

        result := val1 - val2;
        act dist_net.data (<dest1,< result,< awk);
        awk := awk +1;
        act control_net.awk (<awk);
        awk := awk +1;
        act control_net.awk (<awk);
        exit
endproc         /* sub */
```

```
proc div(<format,< opcode,< val1,<val2,<dest1,<awk,<a7,<a8,<a9,<a10);

        if ((format¬= 01) or (opcode ¬= 04)) => do
                print(format);
                print(opcode);
                break(904)
                od
        fi;

        result := val1 / val2;
        act dist_net.data (<dest1,< result, <awk);
        awk := awk +1;
        act control_net.awk (<awk);
        awk := awk +1;
        act control_net.awk (<awk);
        exit
endproc         /* div */

proc mult(<format,< opcode,< val1,<val2,<dest1,<awk,<a7,<a8,<a9,<a10);

        if ((format¬= 01) or (opcode ¬= 03)) => do
                print(format);
                print(opcode);
                break(903)
                od
        fi;

        result := val1 * val2;
        act dist_net.data (<dest1,< result, <awk);
        awk := awk +1;
        act control_net.awk (<awk);
        awk := awk +1;
        act control_net.awk (<awk);
        exit
endproc         /* mult */

proc exp(<format,< opcode,< val1,<val2,<dest1,<awk,<a7,<a8,<a9,<a10);

        if ((format¬= 01) or (opcode ¬= 05)) => do
                print(format);
                print(opcode);
                break(905)
                od
        fi;

        i, result := 1;
        while (i <= val2) do
                result := result * val1;
                i := i + 1
        od;
```

```
        act dist_net.data (<dest1, <result, <awk);
        awk := awk +1;
        act control_net.awk (<awk);
        awk := awk +1;
        act control_net.awk (<awk);
        exit
endproc         /* exp */

proc output(<format,< opcode,< val1,<val2,<dest1,<awk,<a7,<a8,<a9,<a10);

        if ((format¬= 01) or (opcode ¬= 300)) => do
                print(format);
                print(opcode);
                break(930)
                od
        fi;

        result := val1;
        print(result);
        break(999);
        exit

endproc         /* output */

proc ident(<format,< opcode,< val1,<dest1,<dest2,<awk,<a7,<a8,<a9,<a10);

        if ((format¬= 02) or (opcode ¬= 06)) => do
                print(format);
                print(opcode);
                break(906)
                od
        fi;

        act dist_net.data (<dest1, <val1, <awk);
        act dist_net.data (<dest2, <val1, <awk);
        awk := awk + 2;
        act control_net.awk (<awk);
        exit
endproc         /* ident */
```

```
/* deciders: less */

proc less(<format,< opcode,< val1,<val2,<tag1,<dest1,<awk,<a8,<a9,<a10);

        if ((format¬= 03) or (opcode ¬= 17)) => do
                print(format);
                print(opcode);
                break(917)
                od
        fi;

        if (val1 < val2) => result := 1
                else => result := 0
        fi;
        act control_net.packet (<dest1, <result,<tag1, <awk);
        awk := awk +1;
        act control_net.awk (<awk);
        awk := awk +1;
        act control_net.awk (<awk);
        exit
endproc         /* less */


/* boolean operators and control distribution: pass */

proc pass(<format,< opcode,< c1,<tag1,<dest1,<tag2,<dest2,<tag3,<dest3,< awk);

        if ((format¬= 06) or (opcode ¬= 16)) => do
                print(format);
                print(opcode);
                break(916)
                od
        fi;

        result := c1;
        act control_net.packet (<dest1, <result,<tag1, <awk);
        act control_net.packet (<dest2, <result,<tag2, <awk);
        act control_net.packet (<dest3, <result,<tag3, <awk);
        awk := awk +2;
        act control_net.awk (<awk);
        exit
endproc         /* pass */

endmech         /* operation */
```

```
mech dist_net;

        privar dest, result,awk,instr_add       :       seq[31 .. 0] bit;
        privar zero, three, twelve, fifteen     :       seq[31 .. 0] bit;


proc data (<dest,<result,<awk);


        f5[dest] := result;             /* enter value in instruction cell */
        f7[dest] := 1;                  /* set value flag */
        f6[dest] := awk;                /* set awknowlege address */
        instr_add := (dest / 3) * 3;    /* calc instruction address of this operand */
        act memory.enable(<instr_add);  /* activate memory check for enable */
        exit
endproc         /* data */

proc start_up();

        zero := 0;
        three := 3;
        twelve := 12;
        fifteen := 15;

        act memory.enable(<zero);
        act memory.enable(<three);
        act memory.enable(<twelve);
        act memory.enable(<fifteen);
        exit
endproc

endmech
```

```
mech control_net;

        privar dest, result,awk,instr_add,tag,address : seq[31 .. 0] bit;

proc packet(<dest,<result,<tag,<awk);

        if (tag == 20) => do                        /* value packet */
                f5[dest] := result;                 /* set value */
                f7[dest] := 1;                      /* set value flag */
                f6[dest] := awk;                    /* set awk address */
                instr_add := (dest / 3) * 3;        /* calc instruction address of this operand */
                act memory.enable(<instr_add)       /* activate memory check for enable */
                od
        || (tag == 10) => do                        /* flag packet */
                if (result == 0) =>                 /* set gate flag */
                        f3[dest] := 2
                else => f3[dest] := 1
                fi;
                f4[dest] := awk;                    /* set gate awk address */
                instr_add := (dest / 3) * 3;        /* calc instruction address of this operand */
                act memory.enable(<instr_add)       /* activate memory check for enable */
                od
        fi;
        exit
endproc         /* packet */
```

```
proc awk(<address);

        if ( (f6[address] >= 0) and (f6[address] < 1000) ) =>   /* operand has a producer */
                if (f2[address] ¬= 3) =>                                /* operand not a constant */
                        do
                        f3[ f6[address] ] := f3[ f6[address] ] + 1;
                                        /* so increment awk_rec of producing instruction */
                                                                        /* for value operand */
                        act memory.enable (<f6[address])                /* and activate memory check for enable */
                        od
                fi
        fi;


        if ((f4[address] < 0) or (f4[address] >= 1000)) => exit /* gate flag has no producer, so do nothing */
        fi;
        if ((f2[address] > 0) and (f2[address] < 3)) =>         /* if there is a gate flag */
                do
                f3[ f4[address] ] := f3[ f4[address] ] + 1;     /* then increment awk_rec of producing instruction */
                 act memory.enable (<f4[address])               /* and activate memory check for enable */
                od
        fi;
        exit
endproc         /* awk */
endmech         /* control_net */



;init dist_net.start_up()

endsys;         /* mit */
```